Laboratoire de Systèmes Répartis
Ecole Polytechnique Fédérale de Lausanne EPFL

Service SNMP de détection de faute pour des systèmes répartis

Travail Pratique De Diplôme
de
Fabien Reichenbach

Département d'Informatique

Professeur: André SCHIPER
Assistant responsable: Matthias WIESMANN

Février 2002

Table des Matières

Table des illustrations

Liste des tableaux

1 Introduction

La tolérance aux pannes est un des objectif principal pour la mise en oeuvre de systèmes répartis. Les noeuds qui composent un réseau sont de plus en plus diversifiés et nombreux. La progression rapide du nombre de noeuds “wireless” par exemple, possédant une importante puissance de calcul préfigure des réseaux de plus en plus complexes et hétérogènes. Cette évolution induit directement l’accroissement de la complexité des tâches d’administration et de détection des défaillances. Parallèlement, l’avènement de la mise en réseau de millions d’ordinateurs connectés en permanence sur internet offre un immense potentiel pour la création et l’utilisation d’algorithmes répartis. La non-fiabilité des noeuds oblige l’utilisation d’un détecteur de faute qui est intégré dans les implémentations de toolkits de communication réparti.

Ce projet pose la question suivante : est-t-il possible de rationaliser la détection de faute en l’offrant comme un service standard, interopérable et performant en utilisant un standard définit pour l’administration réseau ?

Après avoir précisé les objectifs du projet, le chapitre 1 introduit des notions théoriques de base relatives aux systèmes répartis ainsi qu’une description des aspects de la technologie SNMP utilisée dans ce projet. Le chapitre 2 détaille les besoins et les problèmes à résoudre pour l’implémentation d’un service de détection de faute. Une architecture est ensuite proposée. Basé sur cette architecture, le chapitre 3 décrit l’implémentation du prototype fonctionnel d’un tel service. Ce même chapitre décrit finalement les résultats obtenus par cette implémentation.

1.1 But

La détection de faute est un des aspects important des systèmes répartis et fait en général partie des différents toolkits de communication de groupe. Une autre approche est d'avoir un service de détection de faute indépendant, comme on a des services de noms, de temps

etc... Cette approche impliquerait un standard. Il se trouve qu'il en existe un dans un domaine parallèle. Le standard SNMP (Simple Network Management Protocol) est destiné à aider l'administration réseau et inclut certaines fonctionnalités concernant la détection de faute.

Le but de ce projet est d'explorer les possibilités d'implémentation d'un service de détection de faute en utilisant SNMP, de concevoir et de réaliser un tel service.

1.2 Contexte

Ce projet se situe dans la continuation d'une étude pré-doctorale "Network Aware Failure Detector" par Sirajuddin Shaik Mohammed réalisée au laboratoire de systèmes répartis (LSR) à l'Ecole Polytechnique Fédérale de Lausanne en juillet 2001. Cette étude propose une architecture pour la réalisation d'un détecteur de faute en utilisant le protocole SNMP. Le présent projet propose cependant une autre approche basée sur l'aspect asynchrone de la communication par SNMP.

1.3 Détection de faute et systèmes répartis

Cette section introduit les définitions et concepts relatifs aux réseaux répartis et à la détection de faute. Ces notions devraient permettre au lecteur non averti de comprendre les enjeux de la détection de faute dans des systèmes asynchrones.

Dans le chapitre 2, nous discuterons de la relation qui existe entre ces concepts et le service de détection de faute proposé dans ce projet.

1.3.1 Systèmes répartis

Un système réparti est une collection de composants informatisés connectés par un réseau de communication et reliés logiquement à des degrés divers, que se soit par un système d'exploitation réparti ou des algorithmes répartis. Le réseau de communication peut être géographiquement dispersé (WAN) ou être un réseau local (LAN). Une combinaison des deux est possible. On peut définir un modèle réparti plus précisément selon trois modèles : physique, logique et temporel.

1.3.1.1 Modèle physique

Selon [7] il consiste en un ensemble d'ordinateurs autonomes géographiquement dispersés et interconnectés par un réseau de lignes de communication. Chaque noeud est composé essentiellement par un processeur, une horloge, une mémoire centrale (volatile), une mémoire secondaire (non volatile) et une interface raccordée au réseau de communication. Les noeuds ne partagent aucun composant et ils communiquent exclusivement à travers le réseau de communication.

1.3.1.2 Modèle logique

Ce modèle décrit un système réparti du point de vue des programmes qui s'exécutent sur le système. Il consiste en un ensemble de processus s'exécutant de manière concurrente et coopérants pour réaliser une tâche commune. Un processus représente l'exécution d'un programme sur un noeud du système. Les processus coopèrent en échangeant des messages. Un message est un ensemble d'informations qu'un processus désire communiquer à un ou plusieurs processus du système. Cet échange de messages permet aux processus de prendre des décisions concernant l'état du système. Un message est transmis d'un processus à un autre par le biais d'un canal de communication. Un canal représente une connexion logique entre deux processus.

1.3.1.3 Modèles temporels

On distingue deux modèles de système: un système synchrone et un système asynchrone.

Un **système synchrone** est un système dans lequel (1) il existe une limite connue du délai de transmission d'un message entre deux processus, et (2) il existe une corrélation connue des vitesses relatives entre deux processus. De telles assomptions ne sont pas possibles dans un **système asynchrone** (pas de délai limite dans la transmission des messages, pas de corrélation connue dans les vitesses relatives des processus). Un système asynchrone modélise en particulier un système dont la charge n'est pas prédictible (charge CPU, charge du réseau). C'est le cas de la plupart des systèmes réels utilisés actuellement. Informellement, un système asynchrone est un système où il est impossible de faire la différence entre (1) un délai dans la transmission d'un message et (2) le crash de l'expéditeur de ce message.

1.3.1.4 Processus correct / incorrect

Nous introduisons ici la notion de processus *correct* ou *incorrect* utilisée par la suite pour décrire le "résultat d'impossibilité de FLP" (FLP impossibility result) de Fischer, Lynch, et Paterson [6]. Un processus est dit *correct* s'il ne crash jamais. Un processus qui crash est dit *incorrect*. De plus, un processus crashé le reste pour toujours.

1.3.2 Le problème du consensus

Le problème du consensus est en fait un "dénominateur commun" à de nombreux algorithmes répartis. Sa résolution est une condition nécessaire pour la résolution d'un grand nombre d'autres algorithmes répartis.

De façon informelle, on considère un ensemble de n processus, chaque processus p_i ayant une valeur initiale v_i. Ces n processus doivent se mettre d'accord sur une valeur commune v qui est la valeur initiale d'un des processus. Les conditions suivantes doivent être respectées : tout processus correct doit finir par décider (Terminaison), si un processus décide la valeur v, alors v est une valeur initiale d'un des processus (Validity), deux processus corrects ne peuvent pas décider différemment (Agreement).

1.3.3 Impossibilité du consensus (FLP impossibility result)

Fischer, Lynch et Paterson [6] ont prouvé que dans un système réparti asynchrone, il n'existe pas d'algorithme déterministe qui résolve le problème du consensus lorsqu'un seul processus peut crasher.

1.3.4 Extension du modèle asynchrone

On l'a vu, le problème du consensus n'est pas résoluble dans un système asynchrone. Il est par contre résoluble dans un système synchrone qui est beaucoup trop contraignant. Chandra et Toueg [5] ont cependant montré qu'il est possible de résoudre le consensus en étendant le modèle de système asynchrone par le concept de détecteur de faute.

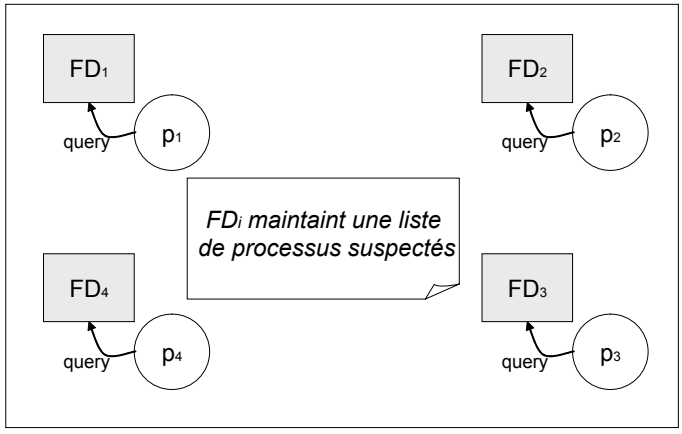
1.3.5 Détecteur de faute

Un détecteur de faute peut être vu comme un oracle qui offre des informations sur des défaillances de processus dans un système réparti.

1.3.6 Propriétés d'un détecteur de faute

- Un détecteur de faute FD_i est un module attaché à un processus p_i. Il maintient une liste de processus suspects.
- Un détecteur de faute peut faire des erreurs en suspectant à tort un processus correct.
- Un détecteur de faute peut changer d'avis. (i.e., suspectant un processus au temps t, et en ne le suspectant plus au temps $t + 1$).
- Des détecteurs de faute peuvent se contredire: à un temps t, un module FD_1 peut suspecter p_k, alors que le module FD_2 ne suspecte pas p_k.

Figure 1-1. Détecteurs de faute



Les détecteurs de faute sont caractérisés par la propriété de *completeness* et la propriété d' *accuracy*. La propriété de *completeness* fait les assomptions concernant les processus incorrects (crashés) alors que la propriété d'*accuracy* fait les assomptions concernant les processus corrects. En voici quelques exemples.

- **Stong completeness** : il existe un instant à partir duquel tout processus défaillant est définitivement suspecté par tout processus correct.
- **Weak completeness** : il existe un instant à partir duquel tout processus défaillant est définitivement suspecté par au moins un processus correct.
- **Strong accuracy** : aucun processus correct n'est jamais suspecté.
- **Weak accuracy** : il existe au moins un processus correct qui n'est jamais suspecté.
- **Eventual strong accuracy** : il existe un instant à partir duquel tout processus correct n'est suspecté par aucun processus correct.
- **Eventual weak accuracy**: il existe un instant à partir duquel au moins un processus correct n'est suspecté par aucun processus correct.

Ces propriétés permettent de caractériser différentes classes de détecteurs de faute.

1.3.7 Classes de détecteurs de faute

On définit différentes classes de détecteurs de faute selon les propriétés de *completeness* et d'*accuracy*. Le *Tableau 1-1* donne ces détecteurs de faute ainsi que leurs notations.

Tableau 1-1. les classes de détecteur de faute

Notation	Satisfait les propriétés
$\mathcal{P}$ *(perfect)*	*strong completeness* et *strong accuracy*
$\Diamond\mathcal{P}$	*strong completeness* et *eventual strong accuracy*
$\Diamond\mathcal{S}$	*strong completeness* et *eventual weak accuracy*
$\Diamond\mathcal{W}$	*weak completeness* et *eventual weak accuracy*

1.3.8 Résolution du consensus avec d$\mathcal{S}$

Le détecteur de faute $\Diamond\mathcal{S}$ est le détecteur le moins contraignant permettant de résoudre le problème du consensus avec l'algorithme proposé par Chandra et Toueg [5].

1.4 SNMP

Ce projet se propose d'explorer les possibilités offertes pas l'utilisation de SNMP, le Simple Network Management Protocol comme élément clé pour la réalisation d'un service de détection de faute. Ce chapitre décrit les principaux concepts et caractéristiques de ce protocole.

1.4.1 Concepts de base

SNMP (Simple Network Management Protocol) définit un ensemble de spécifications pour la gestion de réseau incluant le protocole lui-même, la définition des structures de données, ainsi que les concepts associés. La liste des standards RFC est décrite dans [1] aux pages 75-77. Il est basé sur l'échange de messages entre un élément réseau appelé Agent (ou Serveur) et une station de management appelée NMS (ou Client). Il est le protocole le plus utilisé pour gérer des équipements réseau (routeurs, ponts, etc.) et beaucoup de logiciels de gestion de réseau sont basés sur ce protocole.

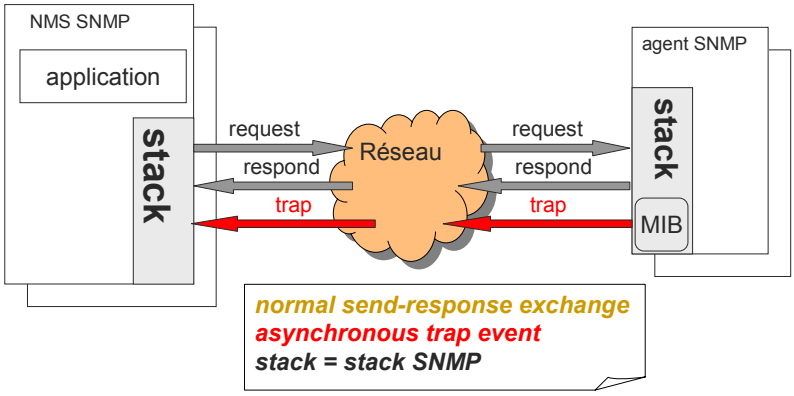
1.4.2 Architecture de SNMP

La mise en œuvre de SNMP dans un réseau nécessite les entités suivantes :

- Station de management NMS (Network Management Station)

- Des éléments de réseaux avec des agents
- Bases de données locales appelées MIB (management information base) maintenues par les agents.

Figure 1-2. Architecture SNMP



1.4.3 Agent

Un agent *(management agent)* est un élément actif du système de management définit par SNMP. Un composant réseau peut être administré par une station de management NMS s'il possède un agent SNMP. L'agent répond aux requêtes de la NMS et peut envoyer de façon asynchrone des messages non sollicités par une NMS. L'agent maintient une base de donnée simple, la MIB qui contient des données relatives au composant réseau.

1.4.4 Manager (NMS)

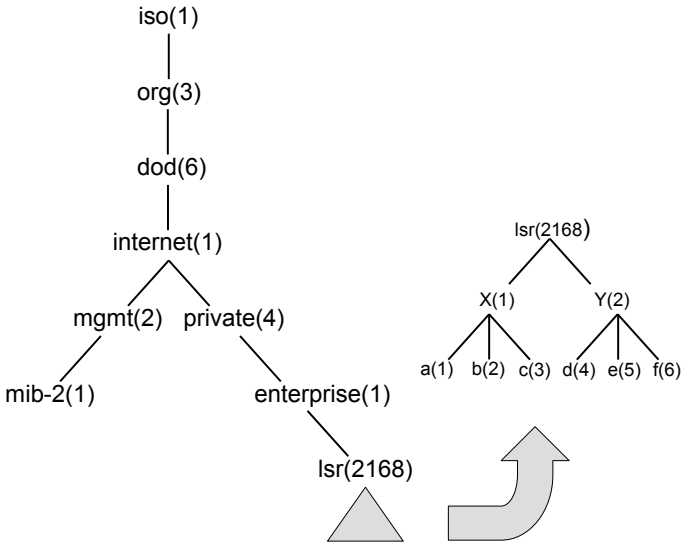
Une station de gestion NMS lit et écrit les valeurs qu'un agent maintient dans une base de données locale appelée MIB. Ces données peuvent être des variables scalaires ou des tableaux bidimensionnels. Une NMS peut aussi recevoir des messages non sollicités d'un agent. Un message non sollicité est appelé trap.

1.4.5 MIB

La MIB est une base de donnée qui ne maintient que des scalaires et des tableaux bidimensionnels. Le framework pour la description et la structure d'une MIB est défini par SMI (Structure of Management Information) spécifié dans RFC 1155. Cette référence définit le langage ASN.1 pour la description d'une MIB, les types de données supportés par une MIB ainsi qu'une méthode d'identification univoque des objets maintenus dans une MIB. Tous les objets gérés via SNMP sont organisés dans une structure arborescente. Les feuilles (et uniquement) représentent des objets qui sont actuellement gérables via SNMP. Chaque

nœud de cet arbre est codé par un entier. On peut donc facilement identifier un objet par la notation pointée **OID** (Object IDentifier) qui décrit les nœuds à parcourir pour atteindre la feuille.

Figure 1-3. structure des oid



Dans l'exemple de la figure 1-3, le *lsr* est identifié par le nœud de valeur 2168 (fictif) de la branche réservée aux entreprises de l'espace d'identifiants SNMP. RFC 1155 stipule que les sous-noeuds de *enterprise(1)* sont attribués par l'*Internet Activities Board*. Les nœuds qui précèdent les feuilles sont appelés des groupes. Chaque groupe peut contenir des valeurs scalaires ou des tableaux bidimensionnels.

Ici l'OID de l'objet d est : **.iso.org.dod.internet.private.enterprise.lsr.Y.d**

équivalent à : **1.3.6.1.4.1.2168.2.4**

En fait il s'agit d'une sorte de constructeur qui représente le type d. On désigne l'instance unique de ce constructeur scalaire en ajoutant .1 à la fin de l'OID. Un type table par contre possède plusieurs instances de type ligne. Pour désigner une table, Asn.1 définit une " SEQUENCE OF object-type-row " ou séquence d'objets de même type ligne. Chaque ligne étant identifiée par un index. Les tables imbriquées ne sont pas supportées.

1.4.6 Les types d'objet

1.4.6.1 Scalaires

Le standard SNMP définit un nombre restreint de types de données scalaires. Les plus importants étant :

Tableau 1-2. les types scalaires

Notation	Satisfait les propriétés
Integer	$(-2^{31} .. 2^{31}-1)$
TimeTicks	entier non-négatif compte le temps en 100ème de sec depuis un temps de référence.
IpAdress	adresse 32 bit utilisant le format d'adresse IP
DisplayString	string de longueur non fixée
OctetString	taille (0..65535)
Object Identifier	SNMP OID, séquence d'entier, lu de gauche à droite, indique la position dans la structure d'arbre de MIB
Gauge	entier non-négatif pouvant s'incrémenter ou se décrémenter[a], avec une valeur max à $2^{32}-1$
Counter	entier non-négatif pouvant uniquement s'incrémenter[b] avec une valeur max de $2^{32}-1$
Opaque	offre la capacité de transmettre des données arbitraires, encodées sous forme de flux d'octet

a. Si la valeur maximum est atteinte, la gauge est bloqué à cette valeur jusqu'au reset.

b. Si la valeur maximum est atteinte, il repart à zéro !!

1.4.6.2 Tableaux bidimensionnels

SNMP offre une deuxième structure de donnée sous forme de tableau bidimensionnel. Une table SNMP contient des objets lignes (row). Chaque objet ligne, contient le même set de types de scalaires. Parmi ces scalaires, un ou plusieurs index sont choisis. Un objet ligne est identifié par un OID unique qui est la concaténation de l'OID de la table avec le(s) index de ligne. De même chaque colonne d'une table est identifiée par un index.

L'accès à un scalaire d'une ligne particulière s'obtient par concaténation des OID de la table, de la ligne et de la colonne (voir [1] section 7.1.3.1 page 161 ou RFC 1155).

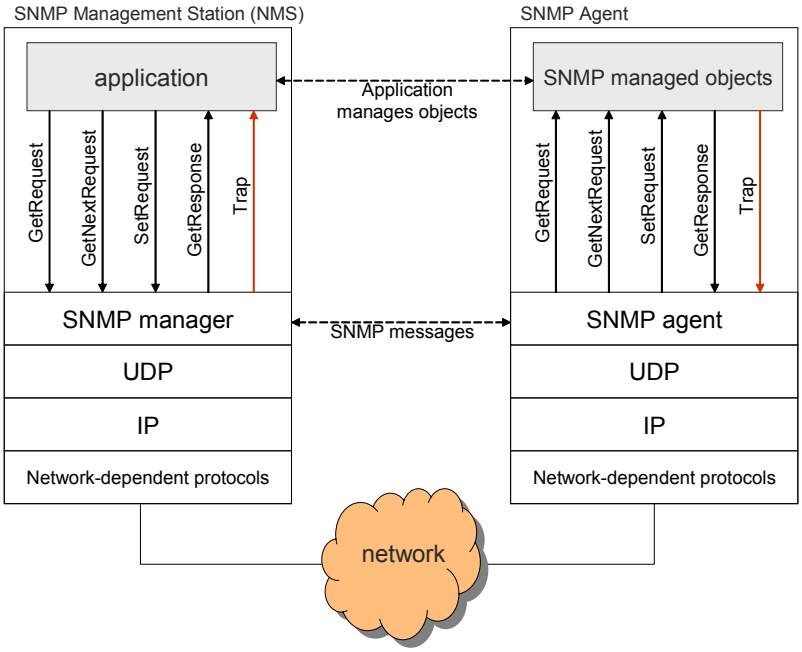
1.4.7 Spécifications du protocole

La communication entre une NMS et un agent peut être synchrone ou asynchrone. Synchrone lorsque un agent renvoie une/plusieurs données contenue(s) dans sa MIB suite à une requête (polling) d'une NMS. Asynchrone lorsqu'un agent envoie spontanément un message (trap) suite à un événement. SNMP est un protocole de niveau applicatif et peut donc

être utilisé via TCP ou UDP. Les messages qui sont envoyés entre une NMS et un agent sont sous la forme de messages SNMP. Ces messages sont composés d'une entête et d'un PDU[1] (Protocol Data Unit). L'entête contient la version de SNMP utilisée et un nom de communauté. Le corps du message contient un des 5 types de PDU. Les différents PDU utilisés pour former les messages SNMP sont décrits dans RFC 1157 disponible à l'adresse *www.faqs.org/rfcs/rfc1157.html*

1.4.8 SNMP formats

Figure 1-4. Rôles et messages dans SNMP



Les PDU échangés sont de trois types : get, set, trap (figure 1-4). Lorsqu'une NMS veut lire un objet contenu dans une MIB, elle envoie un message GetRequest avec l'OID correspondant à l'objet (table ou scalaire). L'agent reçoit la requête et renvoie GetRespond avec la valeur de l'objet ou un code d'erreur si l'objet n'existe pas. Pour mettre à jour un objet, la NMS envoie le message SetRequest avec un OID, l'agent modifie l'objet désigné par l'OID et renvoie GetResponse avec la nouvelle valeur ou un code d'erreur si l'objet est en lecture seule ou absent. Ces messages de confirmation sont nécessaires puisque SNMP a été spécifié pour utiliser avantageusement UDP et donc les messages SNMP peuvent être perdus.

1. La confusion est possible car on parle aussi de PDU pour des packets avec entête.

GetNextRequest est un getter particulier qui permet d'obtenir l'objet "suivant" dans l'ordre lexicographique d'une MIB (voir [1] section 7.1.4 page 164 et Appendix 7A page 197). GetNextRequest est utilisé par une NMS pour découvrir dynamiquement la structure d'une MIB inconnue.

Une trap PDU est envoyé de façon asynchrone (non sollicitée par une NMS) par l'agent lors d'un événement. La figure 1-5 illustre la structure des messages SNMP de type get et set. La figure 1-6 illustre la structure des trap PDU.

1.4.8.1 PDU de type get et set.

Figure 1-5. GetRequest, GetNextRequest, SetRequest et getResponse PDU's format

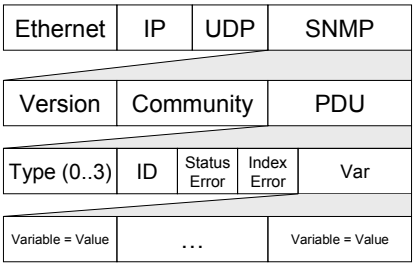


Tableau 1-3. description des champs des messages SNMP (PDU) de type get, set.

Notation	Satisfait les propriétés
Version	SNMP version (RFC 1157)
Community	un nom qui est utilisé comme mot de passe pour des échanges
Type (0..3)	identifie un getRequest, getNextRequest, setRequest ou getResponse
ID	chaque message possède un identifiant unique (corrélation entre request et response)
Status Error	indique le type d'erreur survenu lors du traitement. ex noSuchName = 2
Index Error	associé au status error, permet d'indiquer l'OID de l'objet responsable de l'erreur
Var (Varbind)	une liste de nom de variable et leur valeur correspondante

1.4.8.2 PDU de type trap

Figure 1-6. trap PDU format

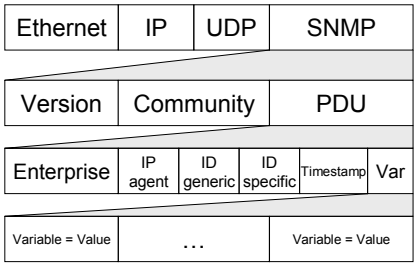


Tableau 1-4. description des champs des messages SNMP (PDU) de type get, set.

Notation	Satisfait les propriétés
Version	SNMP version (RFC 1157)
Community	un nom qui est utilisé comme mot de passe pour des échanges
Enterprise	OID de l'objet qui génère une trap.
IP agent	adresse IP de l'agent qui a envoyé cette trap
ID generic	identifie un type de trap générique[a]
ID specific	identifie un type de trap propre à une implémentation
Time-stamp	temps écoulé entre la dernière réinitialisation de l'agent et l'envoi de la trap.
Var (Varbind)	une liste de nom de variable et leur valeur correspondante

a. les valeurs sont : coldStart(0), warmStartFailure(4), linkDown(2), linkUp(3), anthFailure(4), egpNeighborLoss(5), enterpriseSpecific(6).

1.4.9 MIB standards

Les MIB standards sont un élément central de la spécification SNMP. L'idée est de définir dans un langage de spécification (ASN.1, [1] Appendix B) des MIB spécifiques à des utilisations possibles d'agent. Le but de cette standardisation est d'uniformiser par exemple l'interface de management d'une station de travail unix ou d'un routeur. Ceci indépendamment du matériel et du fabricant.

Des MIB particulières ont été définies pour étendre les fonctionnalités de SNMP. C'est le cas de la MIB RMON qui permet à des NMS d'implémenter des fonctions d'agent. Il devient possible de hiérarchiser les NMS. Exemple : une NMS est informée par des agents que son réseau local sature, elle avertit donc par trap une NMS qui gère l'ensemble des sous-réseaux. Cette dernière configure des routeurs pour désengorger le sous-réseau.

William Stalling [1], chapitre 6, décrit les MIB standards validées par le standard SNMP. RMON et RMON2 sont décrites au chapitre 8 et 10 respectivement.

1.4.10 Versions de SNMP

SNMP évolue et se décline actuellement en trois versions. Les versions 1 et 2 sont proches et garantissent la compatibilité descendante. Nous décrivons ici les particularités propres à chaque version.

1.4.10.1 SNMP V2

- Permet l'implémentation de management centralisé ou réparti. Des NMS peuvent se subordonner à d'autres NMS et ainsi agir à la fois comme station de management et comme agent. Une NMS peut aussi router des traps vers une NMS supérieure.
- Il est possible de créer ou supprimer des lignes de tableaux dans une MIB par l'ajout d'une colonne de type RowStatus. Le protocole de création - suppression est défini dans le livre de William Stallings [1] page 335, paragraphe 11.2.2.3
- Ajout de deux nouveaux PDUs. Le PDU GetBulkRequest permet d'obtenir le contenu d'un tableau entier de manière plus efficace. Le PDU InformRequest permet l'envoi de traps d'un manager à un autre.

1.4.10.2 SNMP V3

- Ajout de fonctionnalités d'encryptage et d'identification relatives à la sécurité.

1.4.11 Limitations intrinsèques de SNMP

- Contrairement à la version SNMPV2, SNMPV1 ne permet pas la modification dynamique du nombre de lignes d'un tableau lors de l'exécution.
- La taille maximale d'un PDU SNMP est limitée mais non fixée par le standard, elle dépend de l'implémentation. Si on envoie un datagramme IP qui est plus large que le MTU (Maximum Transmission Unit) d'un segment du réseau parcouru, il va être fragmenté en plusieurs paquets. Par exemple si on envoie un paquet UDP de 61k, il va être découpé en 42 paquets IP sur un segment Ethernet (1500 bye MTU). La probabilité de perte du PDU entier augmente ici de 42 fois ! Sans compter que la fragmentation et la reconstruction du PDU, est peu efficace. Il y a un autre problème avec la fragmentation : certains fragments se retrouvent sans Header UDP et peuvent être donc bloqués par des firewalls si ceux-ci ne reçoivent pas le paquet contenant le Header UDP en premier. Il est donc important de maintenir des PDU relativement courts.
- Ben-Artzi, Chandna et Warrier [4] montrent que l'acquisition de données par polling (envoi de SetRequest puis attente d'une réponse GetResponse) est peu efficace puisque le nombre de paquets est doublé et que le temps de réponse peu devenir inacceptable .
- SNMP n'est pas bien adapté pour l'acquisition de grands volumes de données, comme une table entière de routage.
- Les traps SNMP sont sans acknowledgements dans le cas de l'utilisation avec UDP, on ne peut pas être sûr qu'une trap est bien arrivée à destination.

2 Un service de détection de faute avec SNMP

Ce chapitre analyse les besoins requis par la notion de service de détection de faute et décrit la couche d'abstraction que cette notion implique. Dans un deuxième temps, les différentes possibilités offertes par le protocole snmp pour la réalisation d'un service de détection de faute sont explorées et analysées. Finalement une architecture du service est proposée.

2.1 Problématique

Les détecteurs de faute sont généralement offerts dans les toolkits de communication de groupe sous forme d'implémentation ad hoc. Avec cette approche, les possibilités de réutilisation sont nulles. Les différents algorithmes possibles doivent être réimplémentés chaque fois. Il n'y a également aucune interopérabilité entre les différents toolkits et de façon plus large avec d'autres applications présentes dans l'environnement réseau. Dans le cas d'un détecteur de faute, il serait intéressant d'obtenir des informations concernant une panne de routeur par exemple. Il est également très difficile de rationaliser l'utilisation des ressources dès lors que chaque toolkit génère ses propres signaux pour implémenter la détection de faute.

Nous cherchons donc à :

- standardiser la détection de faute sous forme de service ouvert
- améliorer les performances en terme de ressource et de qualité de service (QOS)
- implémenter un détecteur de faute qui satisfasse les propriétés $\diamond S$ (section 1.3.7 à la page 6)

2.1.1 Modèle considéré pour le service

- Les processus monitorés par le service de détection de faute sont de type crash-stop (voir section 1.3.1.4 à la page 4). Ces processus sont soit corrects soit incorrects.
- Lorsqu'un agent crash, les processus monitorés sont considérés comme crashés par tous les utilisateurs du service de détection de faute.

- Le service de détection de faute ne détecte pas les fautes byzantines.
- Le service étant indépendant, les caractéristiques propres à un groupe de communication ne concernent pas le service de détection de faute.
- Les canaux de communication utilisés pour les messages SNMP ne sont pas défaillants pour toujours (i.e. il existe un temps t après lequel au moins un message envoyé est finalement reçu). Cette dernière hypothèse est réaliste car en pratique, un réseau n'est saturé que temporairement. La détection de faute dans le cas de défaillances irréversibles du réseau n'a pas beaucoup de sens.

2.1.2 Notations et remarques générales

- Le "Service SNMP de Détection de Faute" est désigné par "SSDF".
- "Toolkit de communication" est pris au sens très large, il peut s'agir par exemple d'un système d'exploitation qui offre des utilitaires de communication répartie. Un toolkit de communication est un utilisateur du service de détection de faute.

2.2 Approche choisie

Ce projet fait le choix de l'utilisation de SNMP pour l'implémentation du service de détection de faute. Cette section traite des options techniques et architecturales prises comme bases pour le design du service.

2.2.1 UDP comme couche de transport

SNMP est un standard de niveau applicatif qui peut très bien utiliser UDP ou TCP. Cette section décrit les arguments qui sont en faveurs du choix d'UDP *(User Datagram Protocol)* pour l'implémentation du service de détection de faute.

2.2.1.1 La tolérance aux pannes

La tolérance aux pannes réseau est une exigence fondamentale pour un service de détection de faute. Dans les réseaux ip où les paquets de données peuvent être routés différemment pour atteindre le même destinataire, il est avantageux d'utiliser des canaux de communication d'hôte à hôte sans connexion comme UDP plutôt que TCP. Si un brin du réseau est surchargé ou physiquement déconnecté, les paquets peuvent être routés par un autre chemin et arrivent finalement à destination.

2.2.1.2 Utilisation de la bande passante et efficacité

L'information échangée dans un détecteur de faute est généralement très simple, elle se résume souvent à un échange de ping. De plus, le service de détection de faute doit générer

un trafic minimum sur le réseau tout en étant efficace. Ces critères font qu'un tel service est avantageusement transporté par UDP. Avec TCP, le maintien de connexions serait coûteux en terme de ressources réseaux et les négociations propres aux protocoles orientés connexion réduiraient la réactivité.

2.2.2 Utilisation du paradigme de SNMP

Un des buts de l'utilisation de SNMP est de rendre possible des interactions entre le service de détection de faute et des applications standards. Pour cela, il est préférable de respecter le framework défini par la spécification SNMP. Comme vu à la section 1.4.2 à la page 6, le système est articulé autour : (1) d'un agent (le serveur) qui possède une MIB, (2) d'un manager NMS (le client). Entre ces deux éléments, une communication basée sur l'échange de messages. L'architecture du service privilégiera une utilisation du "tout SNMP" pour l'implémentation des algorithmes ainsi que la configuration du service.

2.2.2.1 Signification agent / manager

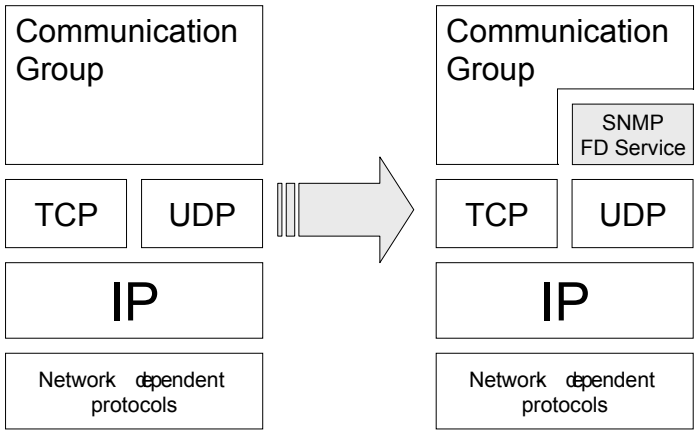
Dans la suite du document : un agent est un serveur au sens d'un agent SNMP. Un manager est un client au sens d'une station de management SNMP. Ces notions sont décrites à la section 1.4.2.

2.3 Architecture

2.3.1 Un modèle en couche

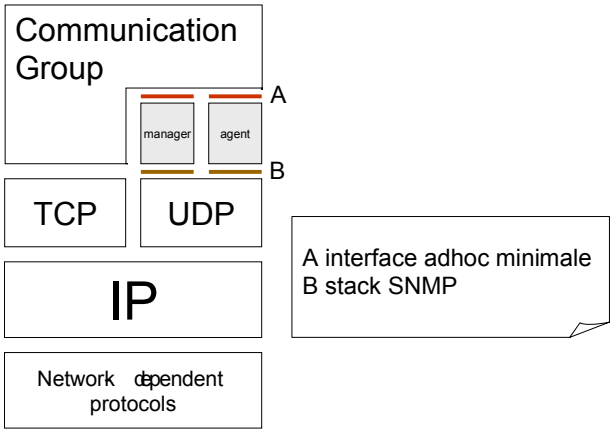
La fonctionnalité de détection de faute est isolée sous forme de couche indépendante. De niveau applicatif, elle se trouve entre la couche de transport UDP et un toolkit de communication de groupe. La figure 2-1 illustre ce modèle.

Figure 2-1. modèle en couche du service de détection de faute



Plus précisément, le service est réalisé par des couples agent-manager comme le montre la figure 2-2. L'agent et le manager possèdent chacun deux interfaces, une interface pour la communication avec le groupe de communication et une interface SNMP pour la communication réseau.

Figure 2-2. modèle en couche détaillé



2.3.2 Description et fonctions de l'agent

Fondamentalement, l'agent offre via une interface SNMP, *une vue partielle* d'un ou plusieurs processus du toolkit de communication répartie. Cette vue partielle est l'état d'exécution d'un processus. Puisque ne sont considérés que les processus de type crash-stop, la vue offerte par l'agent est binaire, soit UP (le processus n'est pas crashé) soit DOWN (le processus est crashé).

2.3.2.1 Monitorable comme vue d'un processus

L'objet de l'agent qui communique avec un processus pour en obtenir l'état d'exécution est un Monitorable. Un objet Monitorable est un *Observable* au sens de *"l'observable design pattern"*, lorsque le processus associé change d'état de fonctionnement, il notifie un *Observer* du nouvel état. L'interface est très simple, une méthode notifyObserver(new-State) qui informe l'observer (l'agent) d'un nouvel état. Un monitorable possède aussi un attribut pour l'état d'exécution du processus et un attribut qui identifie le processus de façon unique. L'attribution d'un nom unique pour un monitorable incombe au toolkit, un doublon génère une exception.

2.3.2.2 Détecteur de crash local

Il faut donc pour utiliser le SSDF implémenter un objet de type Monitorable ad hoc qui puisse s'interfacer avec les processus d'un toolkit de communication particulier. Un tel Monitorable est en fait un *détecteur de crash local* qui, comme son nom l'indique détecte des crashs de processus. Il est important de remarquer que contrairement au détecteur de faute, le détecteur de crash indique l'état d'un processus avec certitude**.**

2.3.2.3 La MIB pour connaître l'état des processus

L'agent maintient dans une MIB une table qui associe à chaque nom de monitorable instancié l'état d'exécution du processus correspondant. Cette MIB permet à n'importe quel outil standard de management SNMP de connaître la liste des processus *monitorés* par un agent à un certain moment et leur état. Elle permet aussi d'implémenter un détecteur de faute par polling successif de l'état d'un processus.

2.3.2.4 Génération de heartbeat

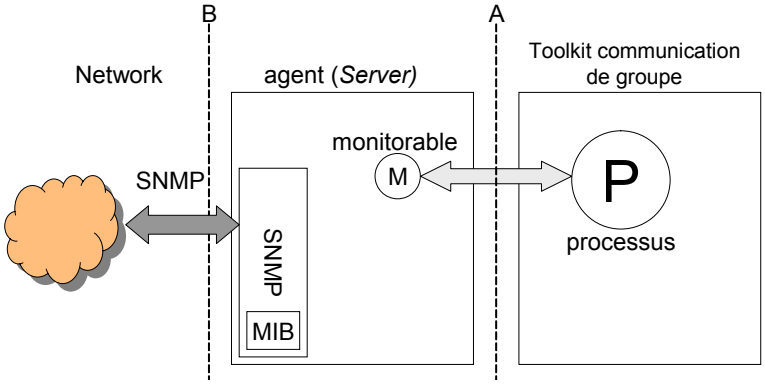
L'agent possède une table dans sa MIB qui permet aux managers de *s'abonner.* Un manager s'abonne à un agent en ajoutant dans une table de l'agent ses coordonnées. Les coordonnées sont au minimum son adresse IP, le port sur lequel il écoute les messages set-get SNMP et le port sur lequel il reçoit les traps. Un manager abonné peut demander à l'agent de lui envoyer des traps à intervalles réguliers, ces traps contenant l'état d'un monitorable. L'envoi régulier de trap par l'agent pour indiquer l'état d'un processus est, dans la suite de ce document nommé *heartbeat d'un monitorable.*

2.3.3 Interfaces de l'agent

Pour réaliser ses fonctions, l'agent offre (A) une interface d'objet monitorable permettant de standardiser la vue de l'état des processus d'un toolkit de communication. Il fournit aussi (B) au réseau une stack SNMP qui offre un accès à une MIB locale.

La figure 2-3, illustre ces deux interfaces.

Figure 2-3. Les interfaces d'un agent



2.3.4 Description et fonctions du manager

2.3.4.1 Module de détection de faute

Le manager est un module de détection de faute particulier. Il maintient une liste de monitorables suspectés d'être crashés et une liste de monitorables dont il est certain qu'ils sont crashés. Le manager diffère du modèle présenté à la section 1.3.5 en ajoutant la liste de monitorables dont il est certain du crash. On peut par contre considérer les monitorables de la liste des monitorables crashés avec certitude comme étant suspectés d'avoir crashés. On ne distingue pas ces deux listes et on les réunit sous le terme "liste de suspicion".

La liste de suspicion est accessible à un toolkit de communication par l'intermédiaire d'un middleware ad hoc qui standardise les méthodes d'accès.

Ce middleware autorise les opérations suivantes:

- obtenir les listes des monitorables suspects (ev. crashés avec certitude)
- interroger l'état d'un monitorable donné connaissant son identifiant
- abonner le manager à un client connaissant son IP et son port
- ajouter un monitorable à suspecter à la liste de monitorables

2.3.4.2 Fonctions SNMP

Le manager possède une stack SNMP qui lui permet de recevoir des traps provenant d'un ou plusieurs agents et il peut envoyer des requêtes SNMP.

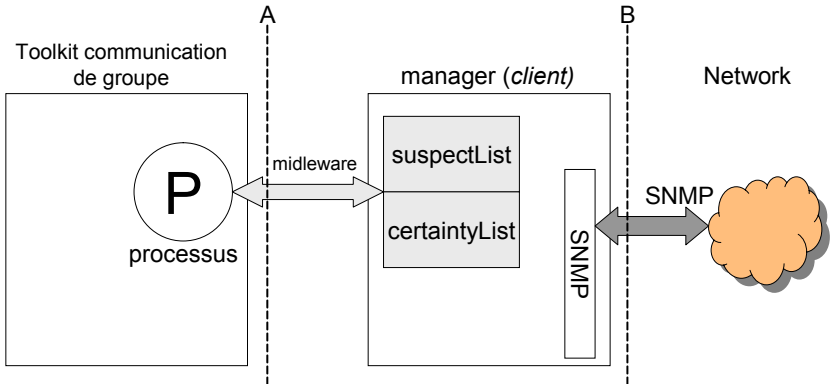
Il peut s'abonner à un agent en ajoutant ses paramètres à la table de l'agent prévue à cet effet. Il peut aussi indiquer à l'agent de quel monitorable il veut recevoir des heartbeats (trap SNMP). Il peut également annuler la notification par heartbeat d'un monitorable ou se désabonner complètement d'un agent.

2.3.5 Interfaces du manager

Pour réaliser ses fonctions, le manager offre (A) une interface de type middleware pour standardiser l'accès aux listes de suspicions par un toolkit de communication. Il fournit aussi (B) au réseau une stack SNMP qui offre un accès à une MIB locale.

La figure 2-3, illustre ces deux interfaces.

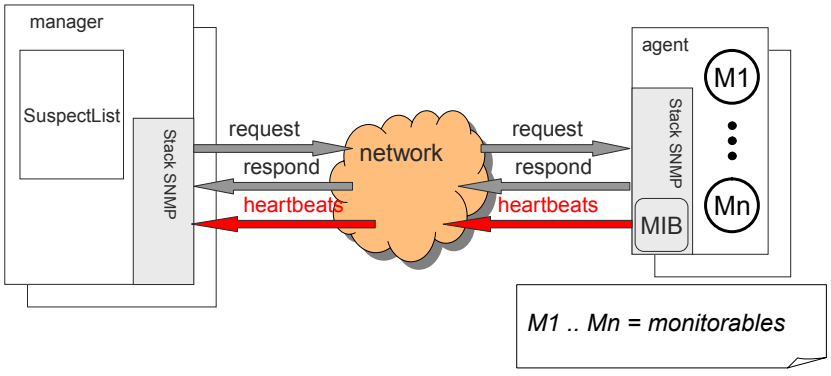
Figure 2-4. Les interfaces d'un manager



2.3.6 Le système Agent-Manager

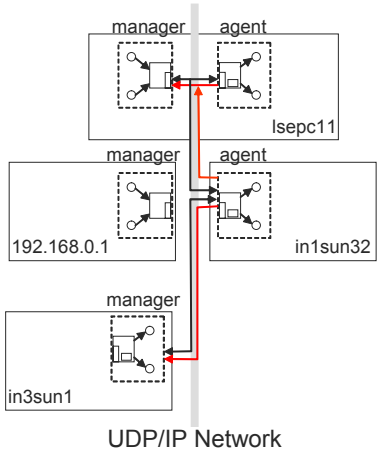
Le service de détection de faute est réalisé par au minimum un agent et un manager.

Figure 2-5. Système Agent - Manager



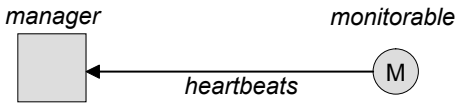
Dans le cas classique où un processus veut pouvoir suspecter un groupe de processus et réciproquement, il faut que les processus d'un site (host) aient accès à un agent et un manager locaux. Mais des configurations non symétriques peuvent être utilisées comme le montre la figure 2-6. Dans cet exemple, deux processus utilisant le service et s'exécutant sur le host lsepc11, peuvent suspecter les monitorables des hosts lsepc11 et in1sun32. Les processus de in3sun peuvent suspecter le monitorables de in1sun32 uniquement.

Figure 2-6. configurations non-symétriques du système



2.4 Détection de faute par heartbeat

Le service réalise sa fonction de détection de faute en implémentant un algorithme simple de heartbeat. En effet, puisque le manager est un module de détection de faute, on peut ramener le système à un ensemble de couple (module de détection de faute, monitorable), le monitorable envoyant des heartbeats à intervalles réguliers au module de détection de faute par l'intermédiaire de l'agent. Les heartbeats sont envoyés sous forme de trap SNMP via UDP et peuvent donc être perdus.



2.4.1 Algorithme

A intervalle régulier, un monitorable *m* envoie un heartbeat au manager *ma.* Lorsque le manager reçoit le heartbeat, il ne suspecte pas *m* d'un crash (TRUST) et démarre un timer

avec une valeur fixée de timeout TO. Si le timer atteint le timeout avant l'arrivée du prochain heartbeat de *m* alors le manager suspecte (SUSPECT) *m*.

Le manager possède donc un timer par monitorable. Dans le cas particulier du SSFD, le monitorable est une vue d'un processus réel et possède un détecteur local de crash. Les heartbeats envoyés par un monitorable sont constitués de l'identifiant du monitorable ainsi que de l'état déterminé par le détecteur de crash. Le manager, lorsqu'il TRUST un monitorable sait si le processus correspondant est crashé (DOWN) ou non.

L'utilisateur du service s'intéresse au processus *p* et non pas au monitorable *m*. L'état *state_p* qu'il obtient sur un processus *p* est alors le suivant:

```
if (manager trust m) then
      if m.heartbeat.state == UP then
            liste.state_p = "TRUST"
      if m.heartbeat.state == DOWN then
            liste.state_p = "DOWN"

else if manager == suspect then
      liste.processus_p = "SUSPECT"
end if
```

Pour l'utilisateur du service, l'état DOWN a la même valeur sémantique qu'un état SUSPECT.

2.4.2 Avantages

L'algorithme est très simple. De plus, le transport de l'information du détecteur de crash permet de raccourcir le temps maximum t_{max} de détection dans le cas où le monitorable n'est pas suspecté. Supposons que le processus *p* crash juste après que son monitorable *m* ait envoyé un heartbeat contenant : *m* is "UP" et soit *d* le délai de ce heartbeat, d_{+1} le délai du heartbeat suivant contenant : *m* is "DOWN" et hb_p la période d'envoi des heartbeats.

le temps maximum de détection est $ta_{max} = hb_p + d_{+1}$

Puisque la période d'envoi des heartbeats hb_p est toujours plus courte que le timeout TO, le SSDF offre un avantage par rapport à une implémentation où c'est le processus lui-même qui envoie directement les heartbeats (on suppose que le délais de détection de crash local $<< hb_p$) .

Dans ce cas, le temps max de détection est $tb_{max} = d + TO$

on a en moyenne : $ta_{max} < tb_{max}$

2.4.3 Désavantages

Cet algorithme possède les faiblesses décrites par Chen et Toueg dans [3], section 1.2. On voit intuitivement que, puisque la réinitialisation plus ou moins rapide du timer dépend du délai de dernier heartbeat *hb*-1, un délai court du *hb-1* augmente la probabilité d'un timeout sur le heartbeat suivant *hb.* Cette dépendance au passé n'est pas désirable. Les auteurs proposent des solutions qui pourraient être implémentées dans le SSDF.

Dans le cas d'un crash d'un monitorable (donc d'un agent), le temps de détection est le même qu'une implémentation où c'est le processus lui-même qui envoie directement les heartbeats.

2.4.4 Optimisation par méta-heartbeat

L'envoi par l'agent, pour chaque monitorable m_i, d'un heartbeat de période p_i hb_i à un manager n'est pas très optimal en terme d'utilisation de la bande passante du réseau. Il est possible de regrouper les heartbeats destinés à un manager en formant un *méta-heartbeat mhb* de période $min(p_i)$ qui est la plus petite période des *hb* envoyés à cet agent. La taille du meta-heartbeat est inférieure aux tailles cumulées des heartbeats individuels puisqu'on évite la multiplication des entêtes de datagrammes UDP.

2.4.4.1 Avantages

Dans le cas où l'agent n'est pas crashé, on diminue encore le temps maximum de détection d'un crash décrit à la section 2.4.2 en raccourcissant hb_p à la plus petite période des *hb.*

dans l'expression $ta_{max} = hb_p + d_{+1}$

Globalement, l'utilisation des méta-heartbeats optimise l'utilisation de la bande passante.

2.4.4.2 Désavantages

La taille des traps est plus importante et l'analyse des méta-heartbeat par le manager consomme plus de ressources CPU.

2.4.5 Perte d'un méta-heartbeat

Globalement, la perte d'un méta-heartbeat n'est pas plus pénalisante que la perte d'un heartbeat individuel, les timers t_i déclanchant une suspicion après un timeout TO_i propre à chaque monitorable indépendamment de la période d'envoi des méta-heartbeat.

2.4.6 Crash d'un agent

Le crash d'un agent amène le détecteur de faute à suspecter définitivement tous les processus monitorés par l'agent.

2.4.7 Crash d'un manager

C'est au toolkit de prendre une décision en tuant par exemple tous les processus associés à ce manager. On pourrait aussi implémenter un mécanisme de redémarrage du manager par le toolkit.

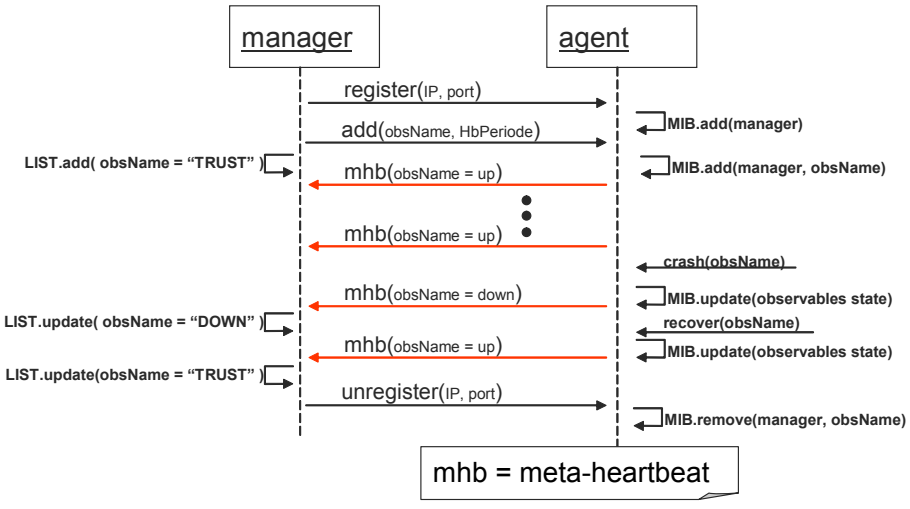
2.5 Scénario d'exécution

prérequis :

- les interfaces agent-toolkit et manager-toolkit on été implémentées par l'utilisateur (i.e les processus à suspecter possèdent leurs monitorables respectifs).
- les agents et managers s'exécutent sur les sites où sont exécutés les processus.

2.5.1 Scénario sans timeout

Figure 2-7. scénario d'exécution A

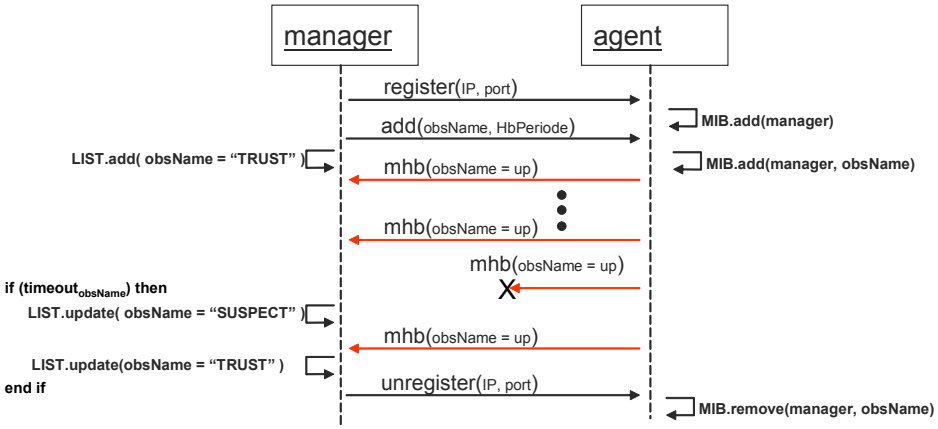


Ce scénario montre les séquences d'opérations entre un agent et un manager lors d'une session où aucune suspicion n'est déclanchée par timeout. *register(IP, port)* représente la

séquence de messages SNMP nécessaire pour ajouter dans la MIB les paramètres qui identifient le manager c-à-d : adresse ip et port SNMP (le port sur lequel le manager écoute les messages SNMP). *add(obsName, HbPeriod)* représente la séquence de messages nécessaire à l'ajout dans la MIB de l'identifiant unique (obsName) du monitorable dont le manager veut recevoir des heartbeats. On peut lire les opérations effectuées par le manager sur la liste de suspicion ainsi que les opérations effectuées par l'agent sur la MIB. Si le manager désire obtenir les heartbeats de plusieurs monitorables d'un agent, il effectue autant de fois l'opération add(obsName, HbPeriod) et les mhb prennent la forme : *mhb(obsName1=State1 ; obsName2=State2; ... ; obsNameN=StateN).*

2.5.2 Scénario avec timeout et perte d'un pdu

Figure 2-8. Scénario avec timeout et perte d'un pdu



Ce scénario illustre le cas d'une perte de meta-heartbeat. Les timers dont les états des monitorables sont transportés par le meta-heartbeat perdu continuent à s'exécuter. Si un timeout est déclanché avant l'arrivée du prochain meta-heartbeat, il y a suspicion du processus.

Remarque **:** ce scénario correspond aussi au cas où un meta-heartbeat (trap) retardé, est reçu après le meta-heartbeat suivant. Dans ce cas, l'agent doit écarter cette trap en se basant sur le timestamp inclu dans chaque trap. La réinitialisation d'un agent (et donc de la référence de temps) implique une réinitialisation des échanges manager-agent. Les mécanismes de recovery ne sont pas pris en compte dans le cadre de ce service prototype.

2.6 Propriétés du détecteur de faute

Nous aimerions montrer que le détecteur de faute est de classe $\diamond S$ (section 1.3.7). Un détecteur de faute de classe $\diamond S$ possède les propriétés de *Strong completeness* et *eventual weak accuracy*. Nous énumérons tous les scénarios d'exécutions possibles afin de montrer qu'ils ne violent pas ces propriétés.

2.6.1 Strong completeness

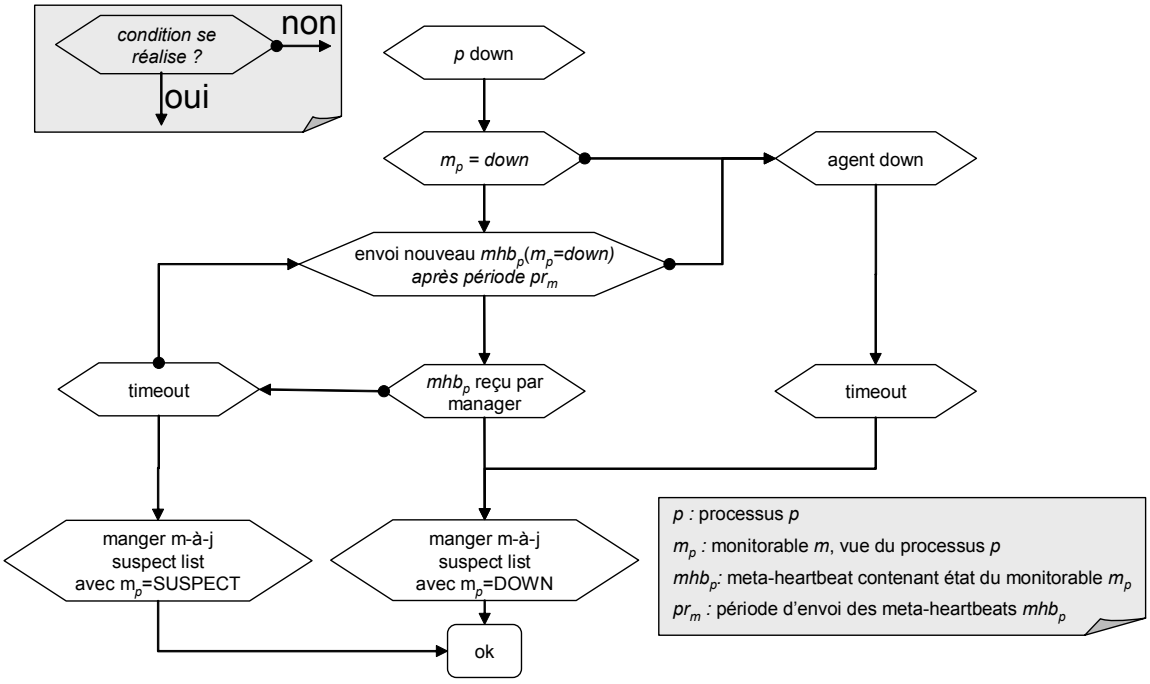
Strong completeness : "il existe un instant à partir duquel tout processus défaillant est définitivement suspecté par tout processus correct".

Le graphe de la figure 2-9 décrit informellement les états possibles du système lorsqu'un processus p crash. On remarque dans tous les cas de figures que la liste de suspicion est finalement mise à jour avec l'état "SUSPECT" ou "DOWN" et ceci tant que p reste crashé ("SUSPECT" ou "DOWN" sont équivalents).

Les états considérés sont :

- ***p down :*** le processus p crash.
- ***m_p = down :*** pour l'agent, le monitorable (vue du processus p) est crashé de façon certaine.
- ***agent down:*** l'agent est crashé.
- ***envoi nouveau $mhb_p(m_p = up)$ après période pr_m :*** l'agent envoie une meta-heartbeat avec l'état du monitorable m_p un temps pr_m après l'envoi du dernier meta-heartbeat.
- **manager m-à-j suspectList avec p = SUSPECT :** La liste de suspicion est mise à jour par le manager avec la valeur p = SUSPECT.
- ***mhb_p reçu par manager:*** Le meta-heartbeat contenant l'état du monitorable p est reçu par le manager.
- ***$timeout_m$:*** Un timer du manager indique que le délai de timeout est passé et que le méta-heartbeat contenant l'état du monitorable m n'a pas été reçu.

Figure 2-9. Strong completeness

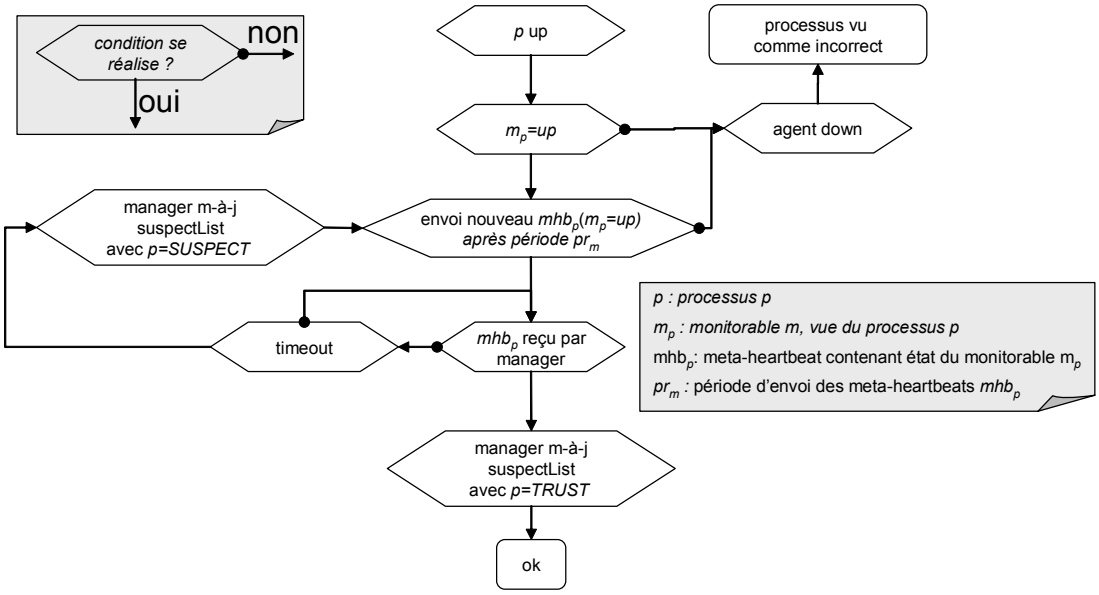


2.6.2 Eventual weak accuracy

"*eventual weak accuracy"* *: il existe un instant à partir duquel au moins un processus correct n'est suspecté par aucun processus correct.*

Le schéma de la figure 2-10 montre que dans le contexte défini par la section 2.1.1, un détecteur de faute (manager) cesse après un certain temps (peut être très long) de suspecter un processus correct. Ceci est valable pour tous les détecteurs de faute du SSDF. Un problème survient lorsqu'un manager crash. Dans ce cas, l'utilisateur du service ne peut plus suspecter les processus. Il faut donc mettre en oeuvre un mécanisme de réinitialisation et se pose alors la question des défaillances du mécanisme de réinitialisation. Les différentes pistes (hardware par exemple) ne sont pas étudiées dans le cadre de ce projet et nous supposons qu'un manager ne crash jamais. Dans ces conditions, le service de détection de faute satisfait $\diamond S$ et peut donc être utilisé pour résoudre le problème du consensus (section 1.3.2).

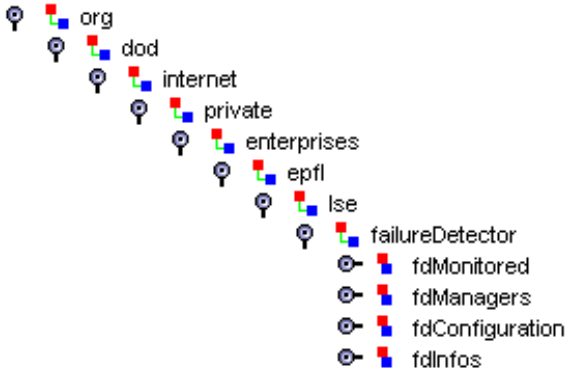
Figure 2-10. eventual weak accuracy



2.7 MIB

La MIB maintenue pas l'agent est structurée en 4 groupes. On se réfèrera au fichier ASN.1 fourni en annexe pour obtenir les détails sur les types de données ainsi que les OID des différents objets.

Figure 2-11. structure globale des oid



2.7.1 Remarques

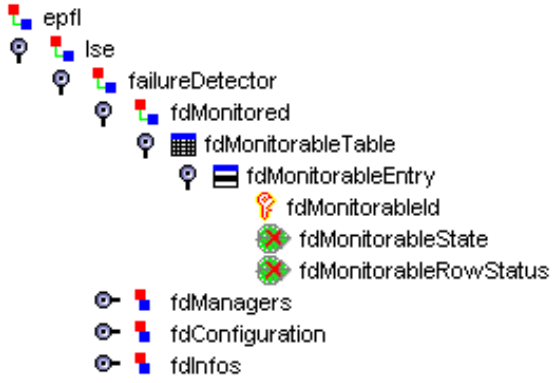
les OIDs actuellement utilisés n'ont pas été enregistrés au près de l'*Internet Activities Board*. Pour une utilisation réelle du SSDF, les OIDs non-enregistrés pourraient entrer en conflit avec des agents SNMP déjà existants.

Dans le cadre de ce projet de faisabilité, nous nous satisferons d'OIDs choisis arbitrairement structurés selon la figure 2-11.

2.7.2 Groupe fdMonitored

Ce groupe regroupe les informations en lecture seule qui concernent les monitorables de l'agent. On y trouve une table bidimensionnelle *fdMonitorableTable* des états des monitorables maintenus par l'agent. Chaque ligne *fdMonitorableEntry* contient un identifiant unique qui identifie le monitorable et sont état associé. N'importe quel manager SNMP standard a donc la possibilité par polling (request-response) de connaître l'état d'un monitorable.

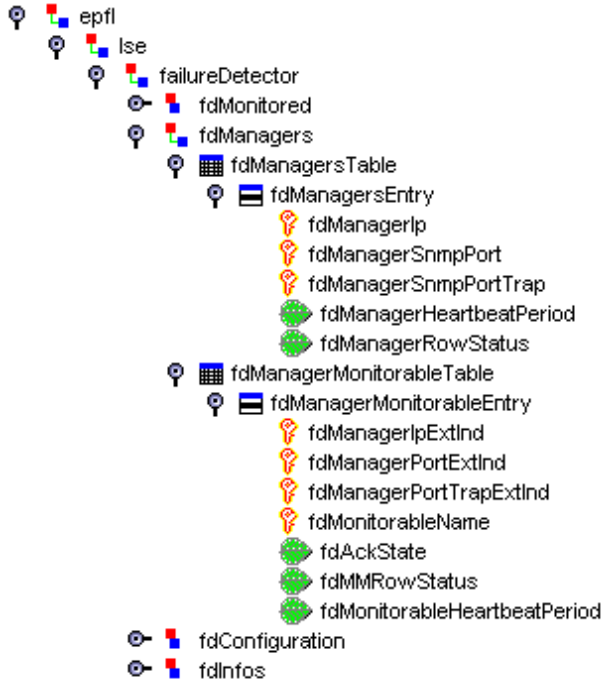
Figure 2-12. structure MIB fdMonitored



2.7.3 Groupe fdManagers

Ce groupe regroupe les informations relatives aux managers. On y trouve deux tables bidimensionnelles. La table fdManagersTable dont les lignes contiennent les informations d'un manager. En y ajoutant ses propres coordonnées, un manager "s'abonne" à l'agent pour recevoir des heartbeats. Un manager est uniquement identifié par le triplet (IP, port SNMP, port trap SNMP). La table fdManagerMonitorableTable et utilisée comme table relationnelle simple. Elle associe aux index d'un manager un identifiant de monitorable. En créant une telle ligne, un manager signifie à l'agent qu'il veut recevoir des heartbeats du monitorable.

Figure 2-13. structure MIB fdManagers



2.7.4 Groupe fdConfiguration

Ce groupe est proposé pour permettre des extensions futures. Il réunirait les objets de configuration dans le cas par exemple où on désirerait configurer par SNMP les paramètres de l'algorithme de détection de faute.

2.7.5 Groupe fdInfo

Egalement proposé pour des développements futurs du service. Ce groupe pourrait maintenir des variables utiles pour des mesures ou statistiques (nombres de heartbeat envoyés par exemple).

2.8 Amélioration de l'algorithme

L'idée est la suivante : lorsqu'un manager reçoit un meta-heartbeat, il confirme, par des messages à l'agent, les états des monitorables reçus dans le meta-heartbeat. Tant qu'un monitorable ne change pas d'état, l'agent enlève celui-ci des meta-heartbeats. Dans un cas idéal, l'agent envoie un meta-heartbeat vide signifiant qu'aucun changement d'état n'est survenu dans l'agent.

2.8.1 Problème de corrélation trap - setRequest

Cette technique pose un problème lié au fait que les traps sont sans relation avec les messages SNMP getRequest et setRequest. Puisque des messages peuvent être perdus ou délivrés dans un ordre arbitraire, l'agent peut prendre en compte un ack qui adressait un changement d'état antérieur.

2.8.2 Solution

Il suffit que le manager envoie avec le message de ack le timestamp de la trap correspondante. L'agent vérifie que le timestamp envoyé est égal ou supérieur à celui de la trap qui a notifié le dernier changement d'état.

On ajoute à la table fdManagerMonitorable (section 2.7.3) les colonnes ack et timestamp. Envoyer un ack pour le monitorable *m* revient à mettre à jour les objets ack et timestamp à la ligne correspondant au monitorable.

2.9 Choix de SNMP V2

La version 2 de SNMP à été choisie parce qu'elle offre un mécanisme de modification dynamique de table bidimensionnelles par un mécanisme de statuts associés à chaque ligne.

3 Implémentation

3.1 Introduction

Ce chapitre expose les détails d'implémentation du service de détection de faute basé sur le protocole SNMP. Avec les commentaires javadoc, cette documentation devrait servir de référence pour permettre la continuation du projet. Une vue globale du système est d'abord présentée puis des diagrammes exposent la structure composant par composant. Ce design est orienté composant. En effet, une totale séparation existe entre les composants qui implémentent l'envoi et la réception de messages SNMP ou "stack SNMP[1]" et les composants qui implémentent l'algorithme de détection de faute. Il est donc facile d'utiliser un toolkit de stack snmp déjà implémenté et de le remplacer selon les besoins. Afin de ne pas réinventer la roue, une étude comparative est effectuée sur les différentes stack snmp disponibles afin d'en réutiliser une.

3.2 Analyse de toolkits SNMP

Un grand nombre de toolkit existe pour développer des applications de management SNMP. Il est par contre plus difficile de trouver des outils de développement d'agent. Ceci s'explique par le fait que les développements logiciels sont plus nombreux du côté manager, les agents existant déjà dans les différents périphériques réseaux. Un bref descriptif est donné pour chaque toolkits. A chaque outil est associé un jeu de critères pondérés selon leur importance positive pour le projet. Les critères ont été pondérés différemments pour l'agent que pour le manager en fonction de l'adéquation des critères avec la fonction désirée.

1. Une "stack snmp" est une implémentation qui réalise les spécifications du protocole SNMP standard. Le plus souvent fournie sous forme d'API.

3.2.1 Critères

Les critères signifient des aspects positifs pour le projet. Ils sont notés de 0 à 10 selon la correspondance de l'outil à ce critère. Dans tous les cas 0 = très défavorable, 10 = très favorable.

- PORT : portabilité, capacité a être multiplateformes.
- SUPPV1: support SNMP V1
- SUPPV2: support SNMP V2
- SUPPV3: support SNMP V3
- DOC : présence et qualité de la documentation fournie
- SOURCE : accès aux sources
- LIC: licence peu contraignante (10 = libre; 0 = propriétaire sans trial)
- ARCH : simplicité de l'architecture
- MANAG : fonctionnalités de management offertes
- AGENT: fonctionnalités d'agent offertes (sauf MIB)
- MIB: fonctionnalités de développement de MIB offertes

3.2.2 Descriptif des toolkits SNMP

3.2.2.1 WILMA

url: ftp://ftp.ldv.e-technik.tu-muenchen.de/dist/WILMA/

Outil SNMP écrit en c, il possède un compilateur de MIB. Ne supporte que la version SNMPV1, licence libre pour éducation et recherche, possède des fonctionnalités agent comme manager ainsi qu'un browser de MIB. Derniers développements en 1995, documentation en partie en allemand.

3.2.2.2 net-snmp

url: sourceforge.net/projects/net-snmp/; net-snmp.sourceforge.net/

Outil SNMP écrit en c et en perl, ne possède pas de compilateur de MIB, possède des fonctions agent avec un générateur de traps. Possède des fonctionnalités de manager et un browser MIB Tk/perl. Licence BSD. Problèmes avec la plateforme win32. SHA authentification et DES encryptage, SNMP V1 V2 V3 supportées.

3.2.2.3 Westhawk

http://www.westhawk.co.uk/resources/snmp/index.html; snmp@westhawk.co.uk

Stack SNMP légère, écrite en java, développée pour des applets manager SNMP. Fournit des exemples et une documentation complète. Le code est commenté. Supporte les 3 versions du protocole. Pas de fonction d'agent mis à part un générateur de trap. La source est inclue avec une licence libre. Des exemples sont fournis et fonctionnent.

3.2.2.4 SNMP support for Perl 5

http://www.switch.ch/misc/leinen/snmp/perl/

Ecrit en Perl, relativement portable et sans modules C à compiler. Ne supporte pas SNMP V3. Pas de fonctionnalités agent sauf l'envoi de trap. Pas d'outil de compillation MIB. Licence libre.

3.2.2.5 AdventNet SNMPAPI

http://www.adventnet.com/products/snmp/index.html

Une entreprise très active dans le domaine SNMP. Offre des toolkits pour agent *AdventNet Agent Toolkit* ou manager *AdventNet Managment Toolkit*. Supporte toute les versions de SNMP. Api's pure Java, Browser MIB et compilateur MIB. Portable à 100%. Très bien documenté et fournit des exemples. Licence commerciale avec période d'essai.

3.2.2.6 SNMP.com Emanate

http://www.snmp.com/products/prodlist.html

Outils de développement d'agent en C avec toutes les fonctionnalités. Licence commerciale sans le code source. Pas de trial version.

3.2.2.7 MG-SOFT SNMP Software Development Lab

http://www.mg-soft.si/agentDesignKit.html

Uniquement plateforme win32, licence commerciale trial 30 jours.

3.2.2.8 AGENT++v3.5

http://www.agentpp.com/agentpp3_5/agentpp3_5.html

Api C++ qui supportent les 3 versions de SNMP. Licence libre, pas de source disponible, l'agent est implémenté par dérivation de classes. Portable (ANSI C++).

3.2.2.9 SNMP++ v3.1

Api C++ orienté management, supporte V1,V2,V3, MD5 et SHA authentification, multi-plateformes.

3.2.2.10 ModularSnmp API's

http://www.teleinfo.uqam.ca/snmp/

Api java, libre d'utilisation, implémente la version V3 du protocole SNMP, permet les fonctions d'agent de manager, envoi de trap. Très mal documenté, complexe à utiliser. Problèmes de compatibilité avec les outils standards. (réception des traps).

3.2.3 Catalogue de solutions

Figure 3-1. tableau des critères pondérés pour un manager

		critères											
	toolkits SNMP	PORT	SUPV1	SUPPV2	SUPPV3	DOC	SOURCE	LIC	ARCH	MANAG	AGENT	MIB	
acteurs	WILMA	1	10	0	0	2	10	10	3	7	7	0	262
	net-snmp	3	10	10	6	3	10	10	6	10	3	0	458
	Westhawk	10	10	10	10	8	10	10	8	10	2	0	580
	SNMP support for Perl 5	10	10	10	0	4	10	10	4	8	3	0	450
	AdventNet SNMPAPI	10	10	10	10	10	5	2	8	10	10	10	588
	SNMP.com Emanate	2	10	10	10	5	0	0	8	8	8	8	398
	MG-SOFT SNMP	1	10	10	10	7	2	0	5	8	8	8	406
	AGENT++v3.5	4	10	10	10	3	0	8	5	8	10	3	404
	SNMP++ v3.1	4	10	10	10	3	0	8	3	8	0	0	366
	ModularSnmp API's	10	0	10	10	2	10	10	4	8	6	0	508
	pondération manager	8	0	9	7	6	8	8	4	10	0	10	

Pour le choix d'un outil pour un manager, les critères de fonctionnalités manager sont pondérées au maximum. Les toolkits qui apparaissent sont AdventNet SNMP API, Westhawk et ModularSnmpAPI's.

Ces trois outils ont été analysés plus en détail le choix s'est porté sur l'Api de Westhawk. Ce toolkit est léger et simple à utiliser pour des requêtes simples. Par contre, il ne gère pas des fonctionnalités de plus haut niveau comme les protocoles de création - destruction de ligne de table SNMP.

Figure 3-2. tableau des critères pondérés pour un agent

		critères											
	toolkits SNMP	PORT	SUPV1	SUPPV2	SUPPV3	DOC	SOURCE	LIC	ARCH	MANAG	AGENT	MIB	
acteurs	WILMA	1	10	0	0	2	10	10	3	7	7	0	262
	net-snmp	3	10	10	6	3	10	10	6	10	3	0	388
	Westhawk	10	10	10	10	8	10	10	8	10	2	0	500
	SNMP support for Perl 5	10	10	10	0	4	10	10	4	8	3	0	400
	AdventNet SNMPAPI	10	10	10	10	10	5	2	8	10	10	10	588
	SNMP.com Emanate	2	10	10	10	5	0	0	8	8	8	8	398
	MG-SOFT SNMP	1	10	10	10	7	2	0	5	8	8	8	406
	AGENT++v3.5	4	10	10	10	3	0	8	5	8	10	3	424
	SNMP++ v3.1	4	10	10	10	3	0	8	3	8	0	0	286
	ModularSnmp API's	10	0	10	10	2	10	10	4	8	6		488
	pondération agent	8	0	9	7	6	8	8	4	0	10	10	

Pour le choix d'un outil pour la partie SNMP de l'agent, on pondère les critères de fonctionnalités agent au maximum. Les toolkits qui apparaissent sont dans l'ordre décroissant AdventNet SNMP API (agent toolkit), Westhawk et ModularSnmpApi.

Le choix initial s'est porté sur ModularSnmpApi pour sa licence libre, mais les difficultés rencontrées notamment à cause du manque de documentation ont amené au choix du toolkit AdventNet pour la MIB et Westhawk pour la génération des traps.

3.3 Détails d'implémentation

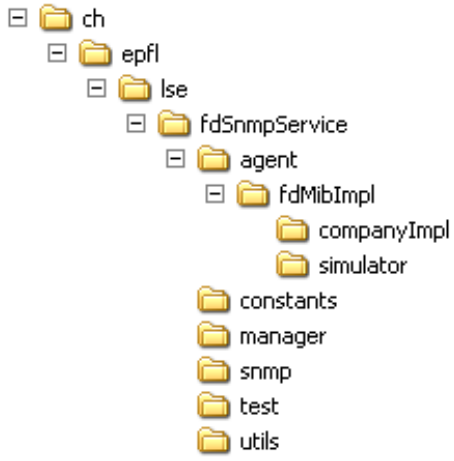
3.4 Package

Le service est délivré sous forme d'un package java structuré selon la figure 3-3. Cette structure permet de séparer les classes propres à un agent de celles propres à un manager.

- ch.epfl.lse.**fdSnmpService** est la racine du package.
- ch.epfl.lse.fdSnmpService.**agent** contient les classes et interfaces formant les divers composants permettant l'implémentation d'un agent. Chaque composant possède une interface.
- ch.epfl.lse.fdSnmpService.agent.**fdMibImpl** contient - pour chaque implémentation préexistante de stack snmp utilisée - un sous-package avec les classes qui réalisent la MIB définie au chapitre 2 (voir fichier de description ASN.1 en annexe) tout en implémentant l'interface FdMibAgent du package agent. De cette façon, on sépare diverses implémentations de stack snmp.
- ch.epfl.fdSnmpService.agent.fdMibImpl.**simulation** contient une version particulière et incomplète d'une MIB puisqu'il s'agit d'un simulateur qui permet de tester le noyau de l'agent sans passer par des messages snmp.
- ch.epfl.lse.fdSnmpService.**constants** contient des interfaces qui définissent les constantes partagées par tout le système (agent et manager). Les messages correspondants à des états (UP ou DOWN) en sont un exemple.

- ch.epfl.lse.fdSnmpService.**manager** contient les classes formant les divers composants qui implémentent un manager. Le sous-package snmpImpl contient les classes qui encapsulent les implémentations existantes de stack snmp tout en implémentant l'interface StackSnmp.
- ch.epfl.lse.fdSnmpService**.snmp** contient des classes relatives à snmp et utilisée par les agents et les managers. Ce package permet une totale indépendance par rapport aux formats de données des packages préexistants.
- ch.epfl.lse.fdSnmpService.**test** contient des classes qui permettent de tester un agent ou un manager avec une configuration prédéfinie.
- ch.epfl.lse.fdSnmpService.**utils** contient des utilitaires communs (entre autre pour le débuggage)

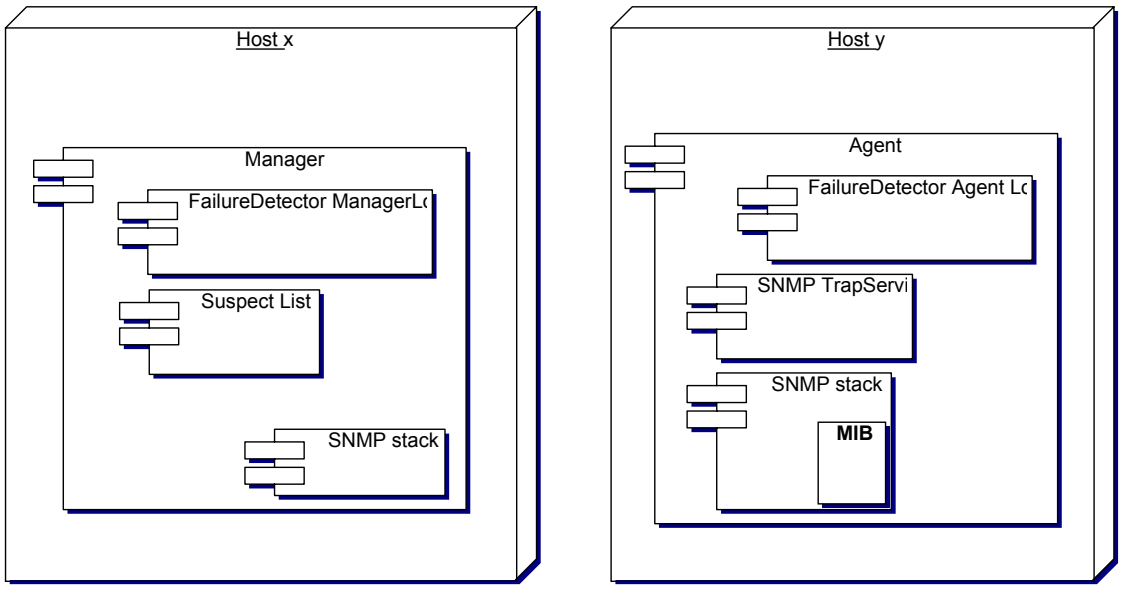
Figure 3-3. package service détecteur de faute



3.5 Composants du système

Comme décrit au chapitre 2, le système est composé de deux entités distinctes. Ce sont deux programmes java qui s'exécutent indépendamment et pas forcément sur le même site. La communication entre ces deux entités est exclusivement réalisée par des échanges de messages SNMP. La figure 3-4 montre le système et ses composants, chaque composant étant défini par une interface rendant ainsi l'évolution et la maintenance du projet la plus aisée possible.

Figure 3-4. vue d'implémentation du Système

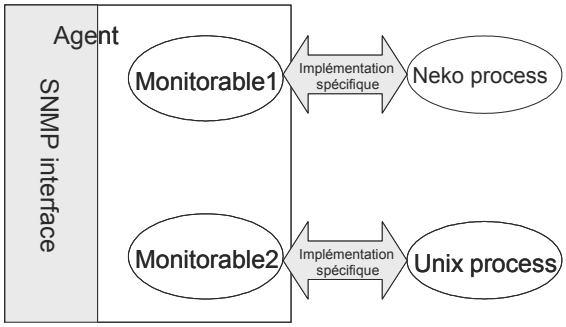


3.6 Agent

L'agent est la partie du système qui interface un ensemble disparate de processus pour en offrir une vue standardisée par SNMP. Pour l'agent, un processus interfacé est un monitorable et peut être assimilé à un proxy de l'état de fonctionnement d'un processus. Il faut donc interfacer des processus d'un côté et d'un autre côté maintenir l'interface SNMP avec leurs états (cf figure 3-5).

Dans l'agent, chaque entité monitorable et représentée par une instance de la classe monitorable. Il s'agit donc pour chaque type de processus d'implémenter une sous-classe de la classe monitorable qui aura la capacité de déceler une défaillance et de la notifier. La notification d'un changement d'état se fait selon le pattern design "Observable" qui associe à un *observable* plusieurs *observer* qui sont notifiés lors d'un changement d'état.

Figure 3-5. Interface d'un Agent avec des monitorables



3.6.1 Composants d'un agent

Les opérations qu'un agent doit effectuer ont été définies et discutées dans le chapitre 2. Nous discutons ici de l'implémentation de ces opérations. De façon générale, un agent doit notifier les changements d'états de ses monitorables aux managers qui se sont "abonnés" au service de détection de faute via la MIB. De plus, un heartbeat doit être envoyé à chaque manager pour qu'ils puissent suspecter l'agent lui-même.

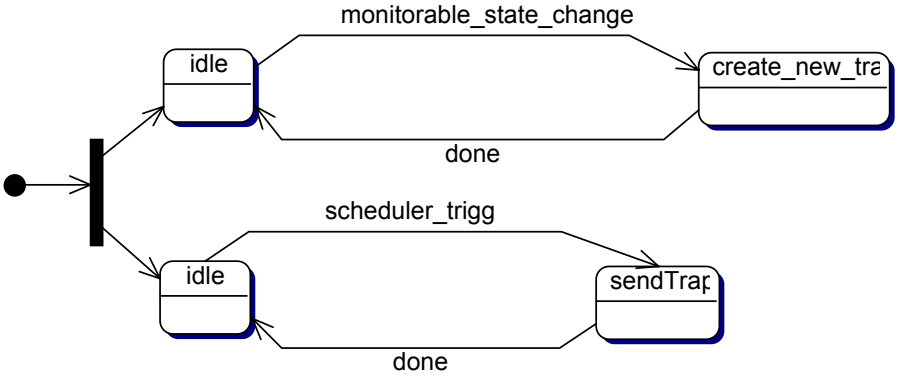
Nous commençons par décrire chaque composant pour ensuite voir comment ils interagissent pour réaliser les opérations de l'agent. Le diagramme de classe de la figure 3-8 illustre les classes importantes de l'agent. Pour plus de détails, se référer aux javadocs du service.

3.6.1.1 trapTask

Une trapTask est un objet actif qui construit et envoie des meta-heartbeats à un manager particulier. Il possède deux listes dynamiques : monitoringList, lookingForList. Ces listes maintiennent les monitorables dont le manager a souscrit et ceux dont il a souscrit mais qui ne sont pas maintenus localement. Cet objet construit le nouveau trap message à envoyer lors d'un changement d'état d'un monitorable qui est maintenu dans les listes monitoringList et ackList. Parallèlement à la construction du nouveau message trap, une trapTask possède un thread bloqué qui est activé par le trapScheduler et qui envoie la trap déjà prête. La figure 3-6 montre les deux activités parallèles de création et d'envoi de message.

Cette structure est efficace puisqu'elle permet l'envoi sans délai des traps ordonnancées par le trapScheduler.

Figure 3-6. Statecharts : trapTask activities



3.6.1.2 heartbeatScheduler

Cet objet est au coeur du dispositif qui permet d'envoyer des meta-heartbeats sous forme de trap aux différents managers. Il schedule l'envoi périodique des traps à chaque manager en activant les trapTask correspondantes (cf "trapTask" à la page 40).

3.6.1.3 trapTaskList

Cet objet est une liste dynamique qui maintient les trapTask du système.

3.6.1.4 SnmpTrapService

L'instance qui implémente cette interface permet l'envoi d'une trap snmp en spécifiant les paramètres de destination (adresse IP, port entre autre).

3.6.1.5 FdMibAgent

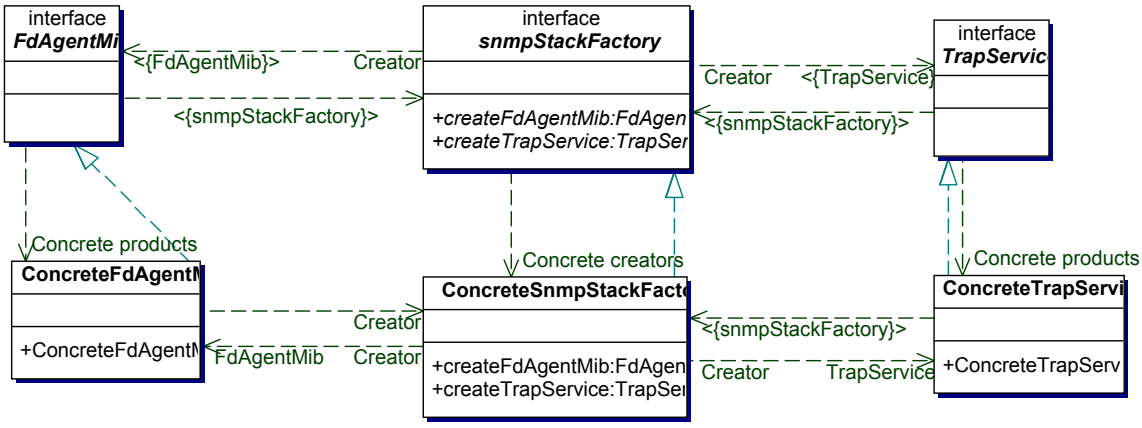
L'instance qui implémente cette interface est l'objet qui maintient la MIB propre au détecteur de faute SNMP définie par le fichier ASN.1 du chapitre 2. Cette interface standardise les mises à jour des états des monitorables dans la MIB par l'agentController. De plus, cet objet implémente la stack snmp et donc traite les requêtes snmp. Lors de l'ajout d'un manager dans la MIB, il notifie l'agentController qui va ensuite scheduler des heartbeats vers ce manager. De même, le FdMibAgent notifie l'agentController lorsque un manager ajoute dans la MIB le nom[1] du monitorable local dont il désire être notifié des changements d'état par heartbeat.

1. Le "nom" d'un monitorable est un identifiant unique propre à ce monitorable.

3.6.1.6 SnmpStackFactory

En utilisant le design pattern “Factory” pour fournir à l’agent l’objet snmpTrapService et le FdMibAgent, il est possible, en ne modifiant qu’un point d’entrée dans le code, de changer l’un ou l’autre des services snmp utilisés par l’agent. Il est donc par exemple très aisé de comparer l’efficacité d’une implémentation de générateur de trap snmp par rapport à une autre.

Figure 3-7. factory design pattern appliqué aux composants snmp de l’agent



3.6.1.7 monitorable

Cet objet abstrait standardise les processus que l’agent monitore localement. Chaque sous-classe de cet objet est implémentée spécifiquement pour chaque type d’entité à monitorer. Un objet monitorable implémente le design pattern “observable”. Il s’agit d’une relation un à plusieurs d’un observable vers un observer, les observers étant notifiés lors d’un changement d’état de l’observable. Dans l’agent, les observers de monitorable sont les trapTasks qui sont notifiées d’un changement d’état et peuvent donc créer un nouveau trap message.

3.6.1.8 monitorableList

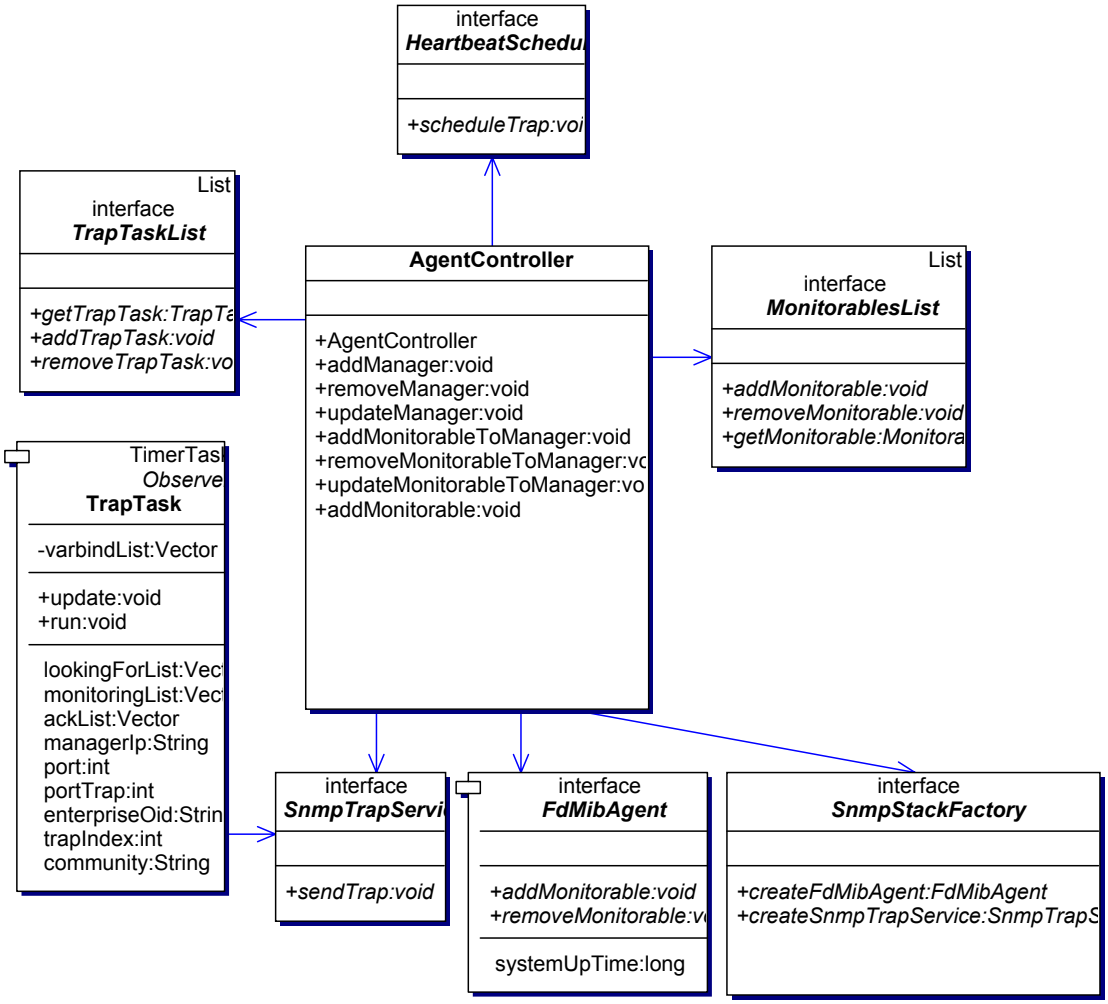
Cet objet est une liste dynamique qui maintient les monitorables locaux de l’agent.

3.6.1.9 agentController

L’agentController est un objet central dont la fonction est de coordonner et contrôler les autres composants. Lorsqu’un manager est ajouté dans la liste des managers de la MIB et qu’ensuite un monitorable y est associé, il crée la trapTask correspondante et démarre le heartbeat associé à l’aide du scheduler. Dès qu’un monitorable est associé à un manager existant dans la MIB, l’agentController vérifie si ce monitorable est bien un monitorable

local à l'agent. Si c'est le cas, il l'ajoute à la liste monitoringList de la trapTask. Les traps suivantes sont alors générées en incluant l'état de ce monitorable. Si, par contre, le monitorable est inconnu de l'agent, il l'ajoute à la liste lookingForList et les traps suivantes indiquent que ce monitorable n'est pas monitoré par l'agent. En résumé, l'agentController est notifié des changements de la MIB et gère en conséquence le mécanisme de génération de heartbeat. Les suppressions dans la MIB sont aussi traîtées. C'est aussi l'agentController qui met à jour les trapTask dans le cas particulier où un monitorable inconnu est "attaché" *a posteriori*.

Figure 3-8. Class diagram : Agent



3.6.2 Génération et ordonnancement des heartbeats

A chaque couple agent-manager contenu dans la MIB de l'agent correspond un meta-heartbeat qui est envoyé périodiquement sous la forme de trap snmp. Ces traps contiennent les noms des monitorables qui ont changé d'états et dont le nouvel état n'a pas été confirmé par le manager[1]. C'est le scheduler qui détermine la période de meta-heartbeat en triggant régulièrement un objet trapTask correspondant à un couple agent-manager. Chaque trapTask maintient les informations qui permettent de générer les traps correspondantes aux spécifications. La figure 3-9 illustre cette architecture.

Figure 3-9. ordonnancement des heartbeat

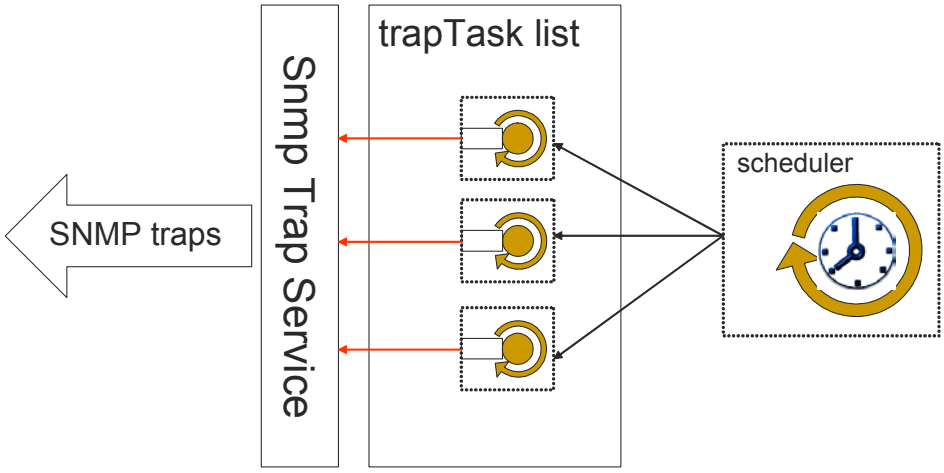
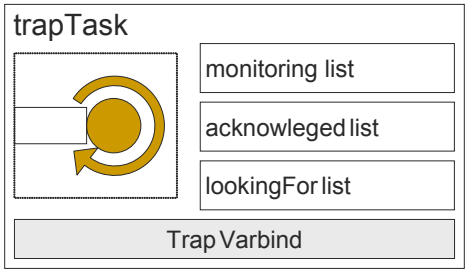


Figure 3-10. structure d'un objet trapTask



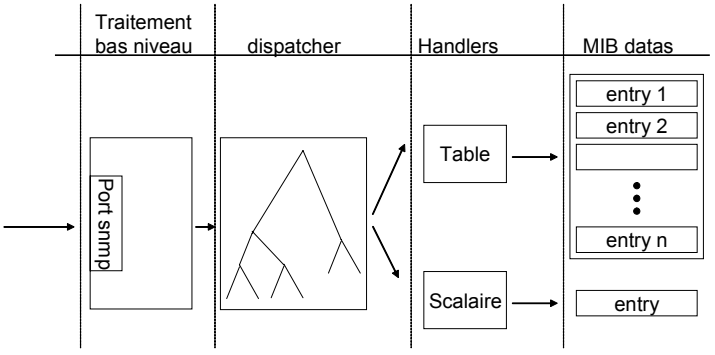
3.6.3 Messages SNMP et MIB

La figure 3-11 illustre la séquence de traitement des messages snmp par la stack snmp ainsi que les mécanismes de gestion des structures de données (scalaire et table) d'une MIB.

1. Uniquement si l'optimisation par ack est implémentée, ce qui n'est pas le cas (encore)

Les messages snmp sont reçus par le port spécifié pour l'agent au démarrage (161 est le port standard snmp). Le traitement des messages est réalisé par trois types de composants distincts. Un composant qui s'occupe de la couche bas niveau du protocole en écoutant sur le port, recevant les paquets udp et vérifiant leur conformité (en-tête, structure des PDU, nb de champs, format etc). Ensuite, les pdu conformes sont envoyés à un dispatcher qui maintient une structure de donnée en arbre organisé selon les OID des objets de la MIB (cf chapitre 5 sur le protocole snmp). A chaque feuille de l'arbre correspond un OID qui identifie un scalaire ou une ligne d'une table de la MIB. S'il s'agit d'un scalaire, le dispatcher redirige la requête (set ou get) vers un handler très simple qui vérifie si une instance de l'objet spécifié (par une oid) existe. Le cas échéant, il transmet la requête à cette instance. S'il s'agit d'une table, c'est un peu plus complexe. La requête est envoyée à un Handler propre à la table qui va mettre à jour une structure dynamique dont chaque objet représente une ligne de la table. C'est ce même Handler qui implémente les protocoles d'ajout et de suppression de ligne défini par SNMP v2. Par instrumentation, on désigne les objets qui implémentent un scalaire ou une ligne de table en fournissant les setter et getter correspondants. L'instrumentation cache, en fait, la façon de maintenir les valeurs (en mémoire ou sur une base de donnée par exemple). Dans notre cas, les variables sont maintenues en mémoire. C'est d'ailleurs à ce niveau que l'on pourrait ajouter la persistance de certaines données.

Figure 3-11. traitement des messages snmp et implémentation MIB



3.6.3.1 Protocoles de création - suppression de lignes

Avec SNMP V2 la possibilité d'ajout-suppression de lignes de tableau est maintenant possible. Cette fonctionnalité a été introduite sans rajout de PDU pour des questions de compatibilités. Pour y arriver, plusieurs protocoles d'échange de message ont été définis. Un

tableau bidimensionnel dynamique doit posséder une colonne spéciale *RowStatus* qui prend plusieurs valeurs qui indique l'état du protocole de création de la ligne. Les algorithmes sont détaillés parWilliam Stalling [1] à la section 11.2.2.3.

CreateAndWait se déroule en plusieurs phases, chaque colonne étant mis à jour successivement. RowStatus est utilisée pour interdire un accès à une ligne en construction dans un état non cohérent (index non initialisés par exemple). Avec cet algorithme, le manager apprend les valeurs par défaut de chaque colonne.

createAndGo permet la création d'une ligne par l'envoi d'un seul message pour autant que le PDU contienne toute les valeurs des colonnes qui n'ont pas de valeurs initiales.

Savoir si un agent est en train de créer une ligne ou de mettre à jour une variable indépendamment n'est pas trivial.

3.6.4 Sections critiques

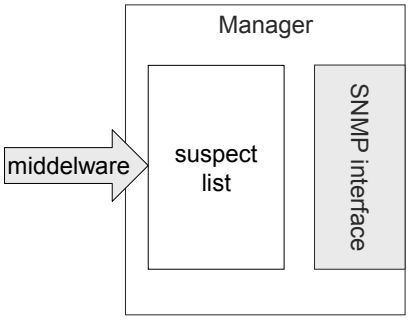
Les instances d'objets HeartbeatScheduler, FdMibAgent, SnmpTrapService, TimerTask, Monitorables, TrapTask, possèdent leur propre thread d'exécution. Les structures de données non thread-safe comme les Iterators utilisés pour créer les traps par exemple sont sécurisés par des méthodes d'accès synchronisées. Chaque cas est décrit par des commentaires javadoc.

3.7 Manager

Le manager est l'oracle du système de détection de faute. Il maintient une liste de monitorables qu'il suspecte ou non d'avoir crashé. Il possède donc deux interfaces, une interface snmp qui lui permet de communiquer avec un ou plusieurs agents et une interface java d'accès à la liste de suspicion. Cette dernière pourrait d'ailleurs être avantageusement remplacée par un middelware tel que Corba ou RMI afin de rendre l'accès à la liste de suspicion facilement accessible à un environnement hétérogène. Dans cette implémentation, nous avons choisi un algorithme classique de détection de faute. Lorsqu'un heartbeat dépasse le délai de timeout pour atteindre le manager, le monitorable est alors suspecté. Si le heartbeat arrive avant l'échéance du timeout, le compteur est réinitialisé et le monitorable cesse d'être suspecté. Il arrive aussi que le heartbeat soit reçu avant le timeout mais indique explicitement que le monitorable correspondant est crashé. Dans ce cas, le monitorable est

indiqué comme étant “DOWN” dans la liste de suspicion. Se référer au chapitre 2 pour plus de détails.

Figure 3-12. Interfaces d’un manager



3.7.1 Notations rappel

Dans les descriptions qui suivent, les traps qu’un manager reçoit d’un agent et qui peuvent contenir l’information de l’état de plusieurs monitorables est appelé *méta-heartbeat*. Un *méta-heartbeat* peut contenir 0,1 ou plusieurs *heartbeats*. Le *heartbeat h* représentant un monitorable *m* est noté *h(m)*.

3.7.2 Les composants d’un Manager

Les composants du manager sont spécifiés par des interfaces java afin d’en faciliter la maintenance et l’évolutivité. De la même manière que pour l’agent, les composants snmp du manager sont encapsulés par des objets implémentant des interfaces standardisées.

3.7.2.1 TrapReceiver

Cet objet écoute sur un port local et reçoit les traps snmp provenant du réseau. Il se charge de les décoder et de les transmettre aux listener timeoutController et snmpManager. Ces traps sont des *méta-heartbeat* envoyés par un ou plusieurs agents.

3.7.2.2 HeartbeatTimer

Cet objet est l’objet actif qui suspecte un monitorablemonitorable *m* s’il ne reçoit pas de heartbeat *h(m)* dans un délai de timeout *TO(m)* défini. Cet objet initialise un timer à chaque réception du heartbeat *h(m)*. Il modifie la suspectList en fonction de l’état du timer à l’arrivée du prochain heartbeat selon l’algorithme défini au chapitre 2.

3.7.2.3 TimeoutController

Le timeoutController reçoit les *méta-heartbeats* (traps) et en extrait les différents *heartbeats* qu'ils contiennent. Il notifie les heartbeatTimer qui correspondent à l'agent émetteur avec les *heartbeats* correspondants. Il maintient donc un index qui associe à l'IP d'un agent une liste de heartbeatTimer. Dans le cas où l'état d'un monitorable a été confirmé à l'agent par un ack(m,state) il n'est plus inclus dans le *méta-heartbeat* tant qu'il ne change pas d'état. C'est le timeoutController qui implémente un mécanisme d'inhibition du heartbeatTimer concerné. Même lorsqu'un heartbeatTimer est inhibé, il est quand même notifié de l'arrivée d'un *méta-heartbeat* et réinitialise son timer sans changer l'état de suspicion du monitorable. Ainsi, l'absence ou le retard d'un *méta-heartbeat* déclanchera aussi la suspicion des heartbeatTimer inhibés à l'échéance de leur timer.

3.7.2.4 SnmpManager

Cet objet gère les requêtes snmp de type set-get envoyées aux agents. Les opérations qu'il peut effectuer sont standardisées par une interface et il est instancié par une snmpFactory. Cette architecture permet d'avoir un seul point d'entrée dans le code à modifier pour changer de composant snmp.

Le SnmpManager est capable de générer les messages snmp de création et de mise à jour des MIB d'agents du détecteur de faute snmp. C'est donc lui qui "abonne" le manager à un ou plusieurs agents dans le but de recevoir des méta-heartbeats. Comme le timeoutController, il implémente l'interface trapListener et est notifié de l'arrivée des nouveaux méta-heartbeats. Dans le cas où l'optimisation est implémentée, la réception des méta-heartbeat lui permet de gérer le mécanisme de confirmation par message ack du changement d'état d'un monitorable en modifiant la MIB de l'agent avec la valeur de ack (voir détails de l'algorithme au chapitre 2). Les ack générés contiennent le timestamp contenu dans la trap reçue.

3.7.2.5 SuspectList

Maintient la liste des monitorables avec leur état de suspicion. Une entrée de cette liste est maintenue par un heartbeatTimer. Les ajouts et suppressions par le SnmpManager.

3.7.2.6 SnmpFactory

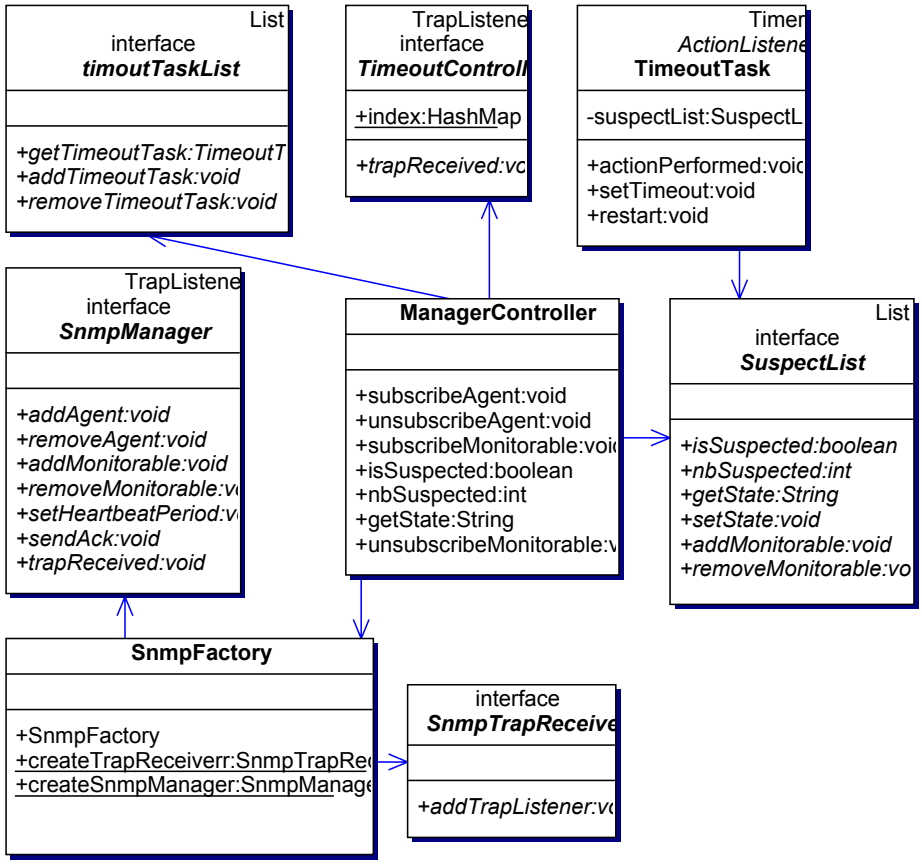
Comme pour l'agent, le design pattern "factory" a été utilisé pour obtenir le trapReceiver et le snmpManager. Cette factory retourne les uniques instances du trapReceiver et du snmpManager, elles-mêmes définies par des interfaces.

3.7.2.7 ManagerController

Le ManagerController est un élément central de contrôle et de coordination du manager. Il offre les interfaces utilisateur pour la consultation de la suspectList ou pour l'enregistrement d'un monitorable sur un agent particulier. Il est aussi responsable de configurer la période des méta-heartbeats pour chaque agent dont il s'est abonné. Pour cela, il choisit la plus petite période de heartbeat des monitorables d'un agent. Il gère aussi les opérations de désabonnement en mettant à jour le timeoutController et la suspectList.

3.7.2.8 Class diagram

Figure 3-13. diagramme de classe du manager



3.8 Résultats

3.8.1 Agent

L'agent fonctionne et implémente la MIB décrite à la Section 2.7. Il permet l'enregistrement d'un ou plusieurs manager et envoie les meta-heartbeats avec le contenu correspondant aux informations ajoutées dynamiquement dans les tables du groupe fdManagers de la MIB. Il est possible d'utiliser un outil standard pour lire les valeurs de la MIB et recevoir les traps. On peut donc simuler le comportement d'un agent avec un outil standard SNMP.

Une Gui minimale de démonstration est implémentée permettant de simuler un crash de processus fictif et d'observer la réaction du système.

3.8.2 Manager

Le manager reçoit les traps, gère les timers et maintient une liste de suspicion. Une Gui minimale permet de voir la liste de monitorables et les états de suspicion qu'elle maintient.

3.8.3 Problèmes rencontrés

Un des problèmes a été la modification concurrente de listes dans le mécanisme de génération de heartbeat.

Les difficultés de mise en oeuvre de la stack ModularSnmpApi et les problèmes de compatibilités de ce toolkit avec les outils standards.

Les problèmes d'algorithmes de création - suppression de lignes tables.

3.8.4 Interopérabilité

Une implémentation d'un monitorable monitorant un processus unix est en cours de développement.

3.8.5 Performance

Aucun test sérieux de performance n'a encore été réalisé. Par contre, le système semble bien supporter la charge en terme de nombre de heartbeats à générer par seconde. Des tests sont en cours de planification.

4 Conclusion

Ce travail de diplôme m'a conduit à essayer de rapprocher deux mondes, celui de l'administration réseau et celui des systèmes répartis. Cette expérience m'aura permis d'expérimenter les difficultés liées au design d'un système, jusqu'a sa réalisation.

Les services de détection de faute basés sur SNMP offrent des possibilités qui semblent prometteuses. J'espère que ce travail servira de base à de futurs projets. En effet, les aspects liés à l'initialisation du système et au bootsrap ont volontairement été laissés de côté pour se concentrer sur la réalisation du socle du service. Ce projet m'a permis de découvrir le standard SNMP et ses diverses facettes.

4.1 Future Works

Dans un premier temps, effectuer un certain nombre de tests pour mesurer les performances globales du système.

L'implémentation de l'algorithme d'optimisation par ack doit encore être réalisée. D'autres méthodes d'optimisation pourraient être testées comme celle qui annule la dépendance au passé discutée par Chen et Toueg [3].

L'aspect de la protection du système aux accès non autorisés n'a pas été traité dans ce projet. Avec SNMP V2, un community name fait office de mot de passe mais il circule en clair sur le réseau. Il serait utile d'utiliser les mécanismes d'authentification supportés par SNMP V3 pour rendre le système de détection de faute moins vulnérable à des modifications non contrôlées de MIB d'agent.

Il serait aussi intéressant d'étudier des extensions au système avec des agents SNMP qui analysent le trafic réseau et qui modifie dynamiquement les caractéristiques du détecteur de faute.

5 Annexes

5.1 Description ASN.1 de la MIB

```
-- File Name : FDAgentMib.mib

FDAgent  DEFINITIONS ::= BEGIN
         IMPORTS
                  DisplayString
                           FROM RFC1213-MIB
                  RowStatus, DisplayString
                           FROM SNMPv2-TC
                  OBJECT-TYPE
                           FROM RFC-1212
                  IpAddress, enterprises
                           FROM RFC1155-SMI
                  enterprises, MODULE-IDENTITY, OBJECT-TYPE, IpAddress
                           FROM SNMPv2-SMI;

         org      OBJECT IDENTIFIER
                  ::=  {  iso  3  }

         dod      OBJECT IDENTIFIER
                  ::=  {  org  6  }

         internet OBJECT IDENTIFIER
                  ::=  {  dod  1  }

         private  OBJECT IDENTIFIER
                  ::=  {  internet  4  }

         enterprisesOBJECT IDENTIFIER
                  ::=  {  private  1  }

         epfl     OBJECT IDENTIFIER
                  ::=  {  enterprises  1008  }

         lse      OBJECT IDENTIFIER
                  ::=  {  epfl  1  }

         failureDetectorOBJECT IDENTIFIER
                  ::=  {  lse  1  }

         fdMonitoredOBJECT IDENTIFIER
                  ::=  {  failureDetector  1  }

         fdManagersOBJECT IDENTIFIER
                  ::=  {  failureDetector  2  }

         fdConfigurationOBJECT IDENTIFIER
                  ::=  {  failureDetector  3  }

         fdInfos  OBJECT IDENTIFIER
                  ::=  {  failureDetector  4  }

         fdMonitorableTableOBJECT-TYPE
                  SYNTAX           SEQUENCE  OF  FdMonitorableEntry
                  ACCESS           not-accessible
                  STATUS           mandatory
                  DESCRIPTION
                           "This Table maintains the list of currently Monitored local Monitorable with
their respecting state"
                  ::=  { fdMonitored  1 }

         fdMonitorableEntryOBJECT-TYPE
```

```
                SYNTAX          FdMonitorableEntry
                ACCESS          not-accessible
                STATUS          mandatory
                DESCRIPTION
                        "A row contains the Monitorable name and its state"
                INDEX           {  fdMonitorableId  }
                ::=  { fdMonitorableTable 1 }

        FdMonitorableEntry  ::=  SEQUENCE {
                fdMonitorableId  Integer32,
                fdMonitorableState  Integer32,
                fdMonitorableRowStatus  RowStatus
                }


        fdMonitorableIdOBJECT-TYPE
                SYNTAX                  Integer32
                ACCESS                  read-only
                STATUS                  mandatory
                DESCRIPTION
                        "This is a unic ID for a particular Monitorable object"
                ::=  {  fdMonitorableEntry  1  }


        fdMonitorableStateOBJECT-TYPE
                SYNTAX                  Integer32
                ACCESS                  read-only
                STATUS                  mandatory
                DESCRIPTION
                        "The actual state of the Monitorable object"
                ::=  {  fdMonitorableEntry  2  }


        fdMonitorableRowStatusOBJECT-TYPE
                SYNTAX                  RowStatus
                ACCESS                  read-only
                STATUS                  mandatory
                DESCRIPTION
                        "Column Description"
                ::=  {  fdMonitorableEntry  3  }

        fdManagersTableOBJECT-TYPE
                SYNTAX          SEQUENCE  OF  FdManagersEntry
                ACCESS          not-accessible
                STATUS          mandatory
                DESCRIPTION
                        "A Manager who register in this table will receive trap heartbeat with Moni-
torable state"
                ::=  { fdManagers  1 }

        fdManagersEntryOBJECT-TYPE
                SYNTAX          FdManagersEntry
                ACCESS          not-accessible
                STATUS          mandatory
                DESCRIPTION
                        "A manager is identified by ip - snmp port - snmp port trap"
                INDEX           {  fdManagerIp, fdManagerSnmpPort, fdManagerSnmpPortTrap  }
                ::=  { fdManagersTable 1 }

        FdManagersEntry  ::=  SEQUENCE {
                fdManagerIp  IpAddress,
                fdManagerSnmpPort  Integer32,
                fdManagerSnmpPortTrap  Integer32,
                fdManagerHeartbeatPeriod  Integer32,
                fdManagerRowStatus  RowStatus
                }


        fdManagerIpOBJECT-TYPE
                SYNTAX                  IpAddress
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "Manager Host Ip"
                ::=  {  fdManagersEntry  1  }


        fdManagerSnmpPortOBJECT-TYPE
                SYNTAX                  Integer32  ( -2147483648 .. 2147483647  )
                ACCESS                  read-write
                STATUS                  mandatory
```

```
                DESCRIPTION
                        "The port the manager wants to receive set-request messages"
                ::=  {  fdManagersEntry  2  }


        fdManagerSnmpPortTrapOBJECT-TYPE
                SYNTAX                  Integer32  ( -2147483648 .. 2147483647  )
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "The port the Agent wants to receive trap messages"
                ::=  {  fdManagersEntry  3  }


        fdManagerHeartbeatPeriodOBJECT-TYPE
                SYNTAX                  Integer32
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "Allow a manager to set the hearbeat period"
                DEFVAL                  { 1000 }
                ::=  {  fdManagersEntry  4  }


        fdManagerRowStatusOBJECT-TYPE
                SYNTAX                  RowStatus  { active ( 1 ) , notInService ( 2 ) , notReady ( 3
) , createAndGo ( 4 ) , createAndWait ( 5 ) , destroy ( 6 ) }
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "Allows dynamic update of the table (MIB-II)"
                ::=  {  fdManagersEntry  5  }

        fdManagerMonitorableTableOBJECT-TYPE
                SYNTAX          SEQUENCE  OF  FdManagerMonitorableEntry
                ACCESS          not-accessible
                STATUS          mandatory
                DESCRIPTION
                        "This Table allow a Manager to suscribe to a particular Monitorable"
                ::=  { fdManagers  2 }

        fdManagerMonitorableEntryOBJECT-TYPE
                SYNTAX          FdManagerMonitorableEntry
                ACCESS          not-accessible
                STATUS          mandatory
                DESCRIPTION
                        "Contains a Manager identifier and a local Monitorable identifier"
                INDEX           {  fdManagerIpExtInd, fdManagerPortExtInd, fdManagerPortTrapExtInd,
fdMonitorableName  }
                ::=  { fdManagerMonitorableTable 1 }

        FdManagerMonitorableEntry  ::=  SEQUENCE {
                fdManagerIpExtInd  IpAddress,
                fdManagerPortExtInd  Integer32,
                fdManagerPortTrapExtInd  Integer32,
                fdMonitorableName  INTEGER,
                fdAckState  DisplayString,
                fdMMRowStatus  RowStatus,
                fdMonitorableHeartbeatPeriod  INTEGER
                }


        fdManagerIpExtIndOBJECT-TYPE
                SYNTAX                  IpAddress
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "The manager IP External Index"
                ::=  {  fdManagerMonitorableEntry  1  }


        fdManagerPortExtIndOBJECT-TYPE
                SYNTAX                  Integer32  ( -2147483648 .. 2147483647  )
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "Manager Snmp port external index"
                ::=  {  fdManagerMonitorableEntry  2  }


        fdManagerPortTrapExtIndOBJECT-TYPE
```

```
                SYNTAX                  Integer32  ( -2147483648 .. 2147483647  )
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "The Manager Port trap external index"
                ::=  {  fdManagerMonitorableEntry  3  }


        fdMonitorableNameOBJECT-TYPE
                SYNTAX                  INTEGER  ( -2147483648 .. 2147483647  )
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "The name (ID) of the local monitorable the Manager wants to get heartbeat"
                ::=  {  fdManagerMonitorableEntry  4  }


        fdAckStateOBJECT-TYPE
                SYNTAX                  DisplayString  ( SIZE ( 0 .. 255  ) )
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "Used by a Manager to acknowledge a new state, this reduce the size of trap
because the Agent remove the corresponding state"
                DEFVAL                  { "UNKNOW" }
                ::=  {  fdManagerMonitorableEntry  5  }


        fdMMRowStatusOBJECT-TYPE
                SYNTAX                  RowStatus  { active ( 1 ) , notInService ( 2 ) , notReady ( 3
) , createAndGo ( 4 ) , createAndWait ( 5 ) , destroy ( 6 ) }
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "allow the dynamic creation of row in this table (MIB-II)"
                ::=  {  fdManagerMonitorableEntry  6  }


        fdMonitorableHeartbeatPeriodOBJECT-TYPE
                SYNTAX                  INTEGER  ( -2147483648 .. 2147483647  )
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "Description"
                ::=  {  fdManagerMonitorableEntry  7  }

        fdProtocolTypeOBJECT-TYPE
                SYNTAX                  Integer32  ( -2147483648 .. 2147483647  )
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "Set the type of protocol to use to implement the fd"
                DEFVAL                  { 1 }
                ::=  {  fdConfiguration  1  }

        fdMaxNbManagersOBJECT-TYPE
                SYNTAX                  Integer32
                ACCESS                  read-write
                STATUS                  mandatory
                DESCRIPTION
                        "Limit the max number of Managers (-1 means no limit)"
                DEFVAL                  { -1 }
                ::=  {  fdConfiguration  2  }

        fdNbOfTrapSentTableOBJECT-TYPE
                SYNTAX          SEQUENCE  OF  FdNbOfTrapSentEntry
                ACCESS          not-accessible
                STATUS          mandatory
                DESCRIPTION
                        "give the number of traps sent by Manager."
                ::=  { fdInfos  1 }

        fdNbOfTrapSentEntryOBJECT-TYPE
                SYNTAX          FdNbOfTrapSentEntry
                ACCESS          not-accessible
                STATUS          mandatory
                INDEX           {  fdInfosManagerIpExtInd, fdInfosManagerPortExtInd, fdInfosManager-
PortExtInd, fdInfosManagerPortTrapExtInd  }
                ::=  { fdNbOfTrapSentTable 1 }

        FdNbOfTrapSentEntry  ::=  SEQUENCE {
```

```
        fdInfosManagerIpExtInd  Integer32,
        fdInfosManagerPortExtInd  Integer32,
        fdInfosManagerPortTrapExtInd  Integer32,
        fdInfoNbOfTraps  Integer32
        }


fdInfosManagerIpExtIndOBJECT-TYPE
        SYNTAX                  Integer32
        ACCESS                  read-only
        STATUS                  mandatory
        DESCRIPTION
                "IP address of a Manager"
        ::=  {  fdNbOfTrapSentEntry  1  }


fdInfosManagerPortExtIndOBJECT-TYPE
        SYNTAX                  Integer32
        ACCESS                  read-only
        STATUS                  mandatory
        DESCRIPTION
                "Manager snmp port"
        ::=  {  fdNbOfTrapSentEntry  2  }


fdInfosManagerPortTrapExtIndOBJECT-TYPE
        SYNTAX                  Integer32  ( -2147483648 .. 2147483647  )
        ACCESS                  read-only
        STATUS                  mandatory
        DESCRIPTION
                "Description"
        REFERENCE        "Manger snmp trap port"
        ::=  {  fdNbOfTrapSentEntry  3  }


fdInfoNbOfTrapsOBJECT-TYPE
        SYNTAX                  Integer32  ( -2147483648 .. 2147483647  )
        ACCESS                  read-only
        STATUS                  mandatory
        DESCRIPTION
                "Column Description"
        ::=  {  fdNbOfTrapSentEntry  4  }

END
```

Bibliography

[1] William Stalling. "SNMP SNMPv2 and RMON" Pratical Network Management Second Edition. Addison-Weslay, January 1997. ISBN 0-201-63479-1

[2] James Rumbaugh, Ivar Jacobson, Grady Booch. "The Unified Modeling Language Reference Manual" Addison-Weslay, December 1998. ISBN 0-201-30998-X

[3] Wei Chen, Sam Toueg, Marcos Kawazoe Aguilera. "On the Quality of Service of Failure Detectors.

[4] Ben-Artzi, A.; Chandna, A.; Warrier, U. (1990). "Network Management of TCP/IP Networks: Present and Future." IEEE Network Magazine, July.

[5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267,March 1996

[6] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382, April 1985.

[7] K.R.MAZOUNI, Etude de l'invocation entre objets dupliqués dans un système tolérant aux fautes, thèse n0 1578 Departement d'informatique Ecole Polytechnique Fédérale de Lausanne , 1996.

[8] V. Hadzilacos and S . Toueg . Distributeds systems , chapter 5 : Fault Tolerant Broadcast and Related Problems , Addison-Wesley , 2nd edition.